

Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master Thesis

Compressed Execution in Hash Joins via Dictionary Vectors in DuckDB

Author: Ahmet Karapinar (2841665)

1st supervisor: Prof. Dr. Peter Boncz

daily supervisor: Paul Groß

2nd reader: Dr. Laurens Kuiper

*A thesis submitted in fulfillment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science*

August 19, 2026

Abstract

An analytical database system holds a query’s working data in memory. That memory is expensive. A query that runs out of it fails or spills to disk. A hash join copies its whole build side into a hash table, which often sets a query’s peak memory. This thesis asks how a hash join can build its hash table in compressed form. Dictionary encoding can compress it, but only if one dictionary covers the whole column. A storage dictionary usually describes one row group rather than the whole column, so storage cannot promise that. A join can. It holds its whole build side before emitting a row, so a dictionary built at that point can cover every value the column takes and never has to change. We call such a dictionary pipeline-global, the main conceptual contribution of this thesis. Three optimizations follow. Small build-side dictionary emission makes a join emit its build-side columns as dictionary vectors, so it produces a pipeline-global dictionary. The dictionary-surviving join preserves a pipeline-global dictionary across the next join’s build side, storing compact indices rather than values. The hash table shrinks, which can lower the query’s peak memory. Dictionary-aware probing resolves each distinct probe key once when the key arrives dictionary-encoded. All three are implemented and merged into DuckDB. On query 7c of the Join Order Benchmark, peak memory fell by 91.1 percent, from 4.3 gigabytes to 385 megabytes. The query ran 2.23 times faster. This is the largest gain we measured across the benchmarks we used.

Acknowledgements

First, I would like to thank Prof. Peter Boncz for giving me the opportunity to work on this project at CWI. It has been a privilege to learn from him. I am equally grateful to my daily supervisor, Paul Groß, who took on this role even while he was doing an internship in Berlin. Over the past six months, the many conversations between the three of us shaped this project and taught me a lot along the way. I owe special thanks to Laurens Kuiper for the time and attention he gave to reviewing all of the DuckDB pull requests behind this work, which is a large part of why the project was completed successfully. Finally, it has been a pleasure to spend this time at CWI among so many excellent researchers.

Contents

1	Introduction	1
1.1	Contributions	4
1.2	Outline	4
2	Background	5
2.1	Database management systems	5
2.1.1	Relational database systems	5
2.1.2	Transactional and analytical workloads	5
2.1.3	Server-based and in-process systems	5
2.1.4	Query processing	6
2.2	Columnar storage	7
2.3	The execution engine	8
2.3.1	Compiled and vectorized execution	8
2.3.2	Vectors and data chunks	8
2.3.3	Pipelines, sources and sinks	9
2.3.4	Logical and physical vector formats	10
2.4	Compressed execution	11
2.5	Dictionary encoding	11
2.6	Dictionaries across consecutive chunks	13
2.7	The hash join	14
2.7.1	The row store and the pointer table	14
2.7.2	The build phase	16
2.7.3	The probe phase	17
2.7.4	Producing the output	17
2.8	Dictionary-aware operators in DuckDB today	18

CONTENTS

3	Literature Study	21
3.1	Compressed execution	21
3.1.1	From storage to execution	21
3.1.2	Direct operation on compressed data	22
3.1.3	The cost of compression-aware operators	22
3.2	Dictionary scope	23
3.2.1	The scopes prior work names	24
3.2.2	Global dictionaries	24
3.2.3	Per-block dictionaries	26
3.2.4	The cost to materializing operators	28
3.3	Dictionary-aware operators	29
3.3.1	Operating on indices instead of values	29
3.3.2	Reusing a resolution across batches	30
3.3.3	Joining on encoded values	31
3.4	Compressing the hash table	31
3.4.1	The hash table as the largest structure	32
3.4.2	Compressing what the table holds	32
3.5	Summary	35
4	Small Build Side Dictionary Emission	39
4.1	The approach	40
4.2	A worked example	42
4.3	Activation	45
4.4	Implementation	47
4.5	Evaluation	48
4.5.1	Setup and methodology	48
4.5.2	Effect of payload width	49
4.5.3	Effect of total rows emitted	50
4.5.4	Worst case	52
4.5.5	Best case	54
5	The Dictionary-Surviving Hash Join	57
5.1	Pipeline-global dictionaries	58
5.2	Deferring the layout decision	61
5.3	The approach	62
5.4	A worked example	63

5.5	Preserving dictionaries through the caching operator	66
5.6	Activation	67
5.7	Implementation	69
5.8	Evaluation	71
5.8.1	Setup and methodology	71
5.8.2	In-memory saving	72
5.8.3	Avoiding a spill	73
5.8.4	Downstream impact	74
6	Dictionary-Aware Probing	77
6.1	The approach	77
6.1.1	Constant keys	79
6.2	A worked example	80
6.3	Activation	82
6.4	Implementation	83
6.5	Evaluation	83
6.5.1	Setup and methodology	83
6.5.2	Probing a pipeline-global dictionary	84
7	Evaluation	87
7.1	Methodology	87
7.2	IMDB	88
7.2.1	The benchmark	88
7.2.2	Activation	89
7.2.3	Results	89
7.2.4	Small build-side dictionary emission	90
7.2.5	The dictionary-surviving join	91
7.2.6	Dictionary-aware probing	95
7.2.7	What the benchmark shows	96
7.3	TPC-H	97
7.3.1	The benchmark	97
7.3.2	Activation	97
7.3.3	Results	97
7.3.4	The dictionary-surviving join	99
7.3.5	The small build-side dictionary emission	100
7.3.6	Dictionary-aware probing	101

CONTENTS

7.3.7	What the benchmark shows	101
7.4	TPC-DS	102
7.4.1	The benchmark	102
7.4.2	Activation	102
7.4.3	Results	103
7.4.4	The small build-side dictionary emission	104
7.4.5	The dictionary-surviving join	104
7.4.6	Dictionary-aware probing	105
7.4.7	What the benchmark shows	105
8	Conclusion and Future Work	107
8.1	Conclusion	107
8.2	Future work	110
8.2.1	More dictionary-aware operators	110
8.2.2	More operators using the pipeline-global guarantee	111
8.2.3	Compressing the key as well as the payload	111
8.2.4	The dictionary-surviving join with storage dictionaries	111
8.2.5	A second dictionary guarantee	111

Introduction

Analytical database systems answer complex queries over large volumes of data. Such queries scan a large part of a table and compute aggregates over it [1], as when a company totals its sales over a year. The system holds such a query’s working data in memory while the query runs, and that memory is expensive [56]. Data analytics workloads need more memory than they used to, and machine memory capacity has not kept pace [27]. Running out of memory is a common form of resource exhaustion in cloud data warehouses, and one of its causes is the class of operators that materialize their input, the hash join among them [58]. Even where a system does not fail, it spills its intermediate data to disk and the query runs slower [56].

Compression is one answer to that problem. It entered database systems as a way to store less on disk [1]. The scan decompresses each column it reads, so every operator above it works on full values. *Compressed execution* is the alternative. The data stays compressed after the scan, and operators work on that form rather than decoding it first [38]. The structures a query builds while it runs then hold compressed values, so the query needs less memory [29].

Many compression schemes exist, and this thesis works on one of them, *dictionary encoding*. It keeps one copy of each distinct value in a small dictionary and replaces every occurrence with a compact integer index. It suits a column such as a country name or a category label, which holds few distinct entries yet appears once per row. Such columns are common. Most columns in the Amazon Redshift fleet repeat their values, as do those in TPC-H and TPC-DS [47]. Across public datasets, over 80 percent of integer columns and 60 percent of string columns repeat each value over a hundred times on average [59].

Dictionary encoding pays most where the values are expensive to handle, and strings are the clearest case. Compared with an integer, a string is expensive for the processor to

1. INTRODUCTION

compare, to hash, and to copy. A materializing operator typically allocates heap memory for every one of them [31]. Studies of real-world workloads, such as Get Real [55], an analysis of the Amazon Redshift fleet [47], and SchemaPile [14], report strings as the most common data type. Operating on the encoded form can cut these costs [29, 3].

This thesis focuses on the hash join. It materializes one of its two inputs, called the build side, into a hash table. That table holds every build row, so it is often large and can set a query’s peak memory [31]. Building it in compressed form can lower the memory a query needs, and that is the main goal of this thesis.

Dictionary encoding can compress that table, if the dictionary has the right scope. The join pins one dictionary and stores an index into it in every row, so every batch has to carry the same dictionary, one that covers the whole column. A storage dictionary usually describes one row group or data block rather than the whole column, so storage cannot promise that [8].

The output of a join is one place such a dictionary can come from. When a small dimension table is joined to a much larger fact table, as in a star schema [36], the join copies the small side’s values into every matching output row. A column with a few hundred distinct values can be copied out tens of millions of times. That is exactly the shape a dictionary describes, a small set of values repeated across many rows. The join could emit the column dictionary-encoded instead, avoiding redundant copies and handing the dictionary to the operators that follow.

The dictionary such a join builds covers the whole of the column it emits. The join has its whole build side in hand before it emits a row, so one dictionary describes every value the column takes and every batch carries that same dictionary. That is the guarantee the build side needs. We introduce a new term for it, and call a dictionary *pipeline-global* when its producer uses the same dictionary, under one stable identifier, on every batch for as long as it runs. The pipeline-global dictionary is the main conceptual contribution of this thesis. Prior work uses column-wide for a dictionary over a stored column [38]. A pipeline-global dictionary is instead built by an operator and held for that operator’s lifetime. A storage dictionary changes from one block of stored rows to the next, and a local dictionary describes only its own batch. A pipeline-global dictionary never changes, so a consumer can commit to it on the first batch. The small build-side dictionary emission this thesis adds produces one.

The emitted column can flow straight into the build side of the next join. To build its hash table, the join writes every build row into a row store, storing each column at its full width, one value per row. The dictionary is discarded, and the same value is stored once

for every row that uses it. Since the dictionary is pipeline-global, the join can pin it on the first batch and every later index still resolves. The join could store compact indices in the row store instead. The table is then smaller, which can lower the query’s peak memory, and the dictionary survives into its output.

A further opportunity is the probe key, the value a join looks up in its hash table. When it arrives dictionary-encoded, only a few distinct values appear across all the rows, yet the join looks up every row on its own. It could resolve each distinct value once and reuse the result.

We study this in DuckDB, an open-source analytical database system [45]. DuckDB executes queries over *vectorized batches* and represents a dictionary-encoded column as a dictionary vector. Other vectorized engines carry a similar representation, among them Velox [54] and ClickHouse [49]. The problem is not specific to DuckDB. It arises in any engine that carries dictionary-encoded columns through its operators.

Within the scope of this thesis we implement each of these opportunities in DuckDB’s engine. A column can then stay encoded across a chain of joins, and later operators can work on the compressed form as well. On query 7c of the Join Order Benchmark [37], whose data is real rather than generated, peak memory fell from 4.3 gigabytes to 385 megabytes and the query ran 2.23 times faster. These are the largest savings in the suite, in both memory and time.

This thesis is guided by the following research questions.

RQ1. How can a hash join produce dictionary-encoded output rather than materializing build-side values for every row?

RQ2. How can a dictionary-encoded column be carried through the build side of a hash join in its compressed form?

RQ3. How can a hash join exploit a dictionary-encoded probe key to resolve each distinct value only once?

RQ4. How much code change do these optimizations require in DuckDB, and is that footprint small enough for the change to be maintainable in a production engine?

RQ5. What effect do these optimizations have on analytical query execution?

RQ5.1. What is their effect on query performance?

RQ5.2. What is their effect on memory use?

RQ5.3. What is their effect on the operators downstream of the join?

1. INTRODUCTION

1.1 Contributions

We make the following contributions.

- **Literature study.** We survey compressed execution, the scopes a dictionary can have, what an operator can do with one, and how the field has compressed the structure a hash join builds.
- **Small build-side dictionary emission.** We make a hash join emit its build-side output columns as dictionary vectors, rather than copying values row by row, so it hands a pipeline-global dictionary to the operators that follow.¹
- **The dictionary-surviving join.** We introduce the pipeline-global dictionary and make a hash join preserve one across its build side, storing compact indices rather than values. A dictionary produced by one join then survives the build of the next, which shrinks the hash table and can lower a query's peak memory.²
- **Dictionary-aware probing.** When the probe key arrives dictionary-encoded, we make a hash join resolve each distinct value once and reuse the result for every row that shares it.³

All three optimizations are merged into DuckDB.

1.2 Outline

Chapter 2 gives the background, from columnar storage to vectorized execution and the hash join. Chapter 3 places this thesis against work on compressed execution, on what a dictionary's scope permits, on what operators do with a dictionary, and on compressing the hash table.

Chapter 4 covers the small build-side dictionary emission, Chapter 5 the dictionary-surviving join, and Chapter 6 dictionary-aware probing. Each gives the design, a worked example, the conditions under which the optimization runs, where the change sits in the code, and a measurement in isolation. Chapter 7 runs all three together on the Join Order Benchmark, TPC-H and TPC-DS. Chapter 8 answers the research questions and discusses future work.

¹<https://github.com/duckdb/duckdb/pull/22340>

²<https://github.com/duckdb/duckdb/pull/23360>

³<https://github.com/duckdb/duckdb/pull/22789>

2

Background

2.1 Database management systems

2.1.1 Relational database systems

A database management system stores data and answers queries over it. Applications reach it through a declarative language, usually SQL. The advantage is that a query says which data is wanted without saying how to find it, so the system is free to choose that for itself. Codd made that separation the point of his relational model, arguing that a user should not need to know how the data is laid out inside the machine [13]. The model organizes data into tables of rows and columns, and connects those tables through keys. A join is the operation that follows those connections.

2.1.2 Transactional and analytical workloads

Database systems are built for different workloads. Two broad classes matter here. Online transaction processing (OLTP) handles many short operations. Each one reads or writes a small number of rows, such as recording an order. Online analytical processing (OLAP) answers fewer queries, but larger ones. Such queries scan a large fraction of a table and compute aggregates over it. They also touch only a few of that table's columns rather than all of them [1]. Computing sales totals across a year is a typical example.

2.1.3 Server-based and in-process systems

Systems also differ in how they are deployed. A server-based system runs as its own process, and clients connect to it over a network. An in-process system is a library instead, running inside the program that uses it. SQLite is the most widely deployed example, built

2. BACKGROUND

for transactional work [25]. Raasveldt and Mühleisen had already shown demand for in-process analytics with MonetDBLite, which was derived from the MonetDB system rather than built for the purpose. They observe that no purpose-built embeddable analytical database existed, and built DuckDB to fill that gap [45]. DuckDB is the system this thesis extends. A program links it in and calls it directly. In 2019 it had no server process at all [45]. A remote protocol called Quack has since been added, still in beta, so DuckDB can now also run as a server [15].

2.1.4 Query processing

Query processing is the work a system does between receiving a query and returning its result. It parses the text, binds the names in it, plans how to compute the answer, then executes that plan [38]. Systems like DuckDB follow the textbook separation into a parser, a logical planner, an optimizer, a physical planner and an execution engine [45]. Figure 2.1 traces a query from SQL text to result and names the form it takes at each step.

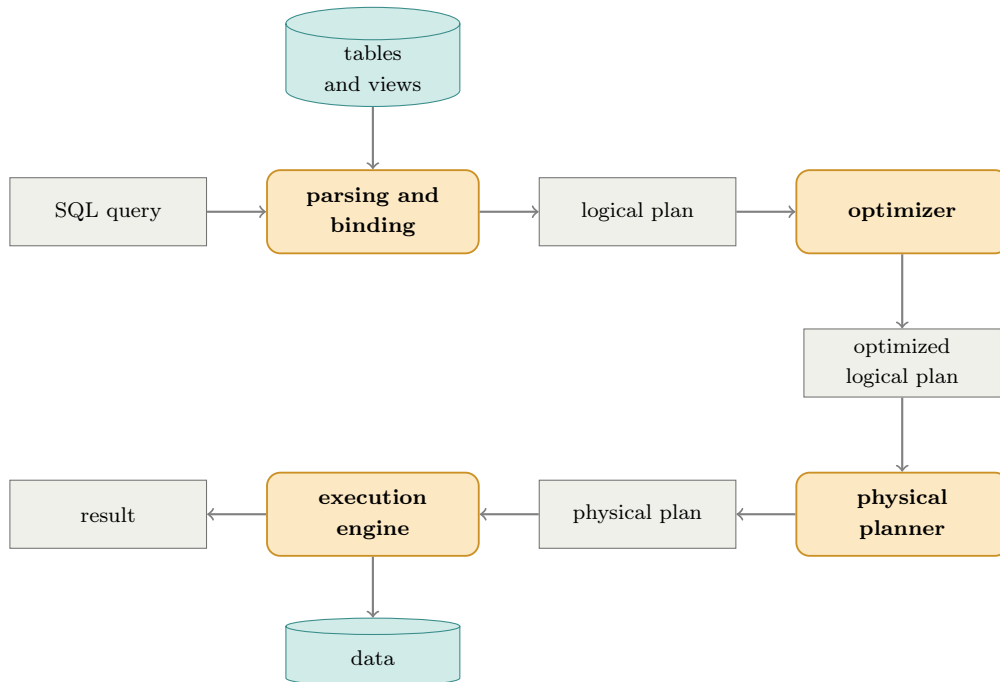


Figure 2.1: The steps of query processing.

Parsing reads the SQL query and returns a parse tree. Binding resolves the names in that tree, matching each reference to a table or a view against that object’s column names and types. A plan generator then turns the bound tree into logical operators such as scan, filter and project, which form the logical plan [45]. The optimizer rewrites that

into a cheaper equivalent, the optimized logical plan, by reordering joins and simplifying expressions [45]. The physical planner then chooses an implementation for each operator, giving the physical plan. A logical operator names a result, while a physical operator is an algorithm that computes one [28]. The execution engine runs those algorithms over the data and returns the result [45].

A hash join and a nested-loop join are two implementations of the same logical join. The hash join is the one this thesis works on. The data it reads is laid out by column.

2.2 Columnar storage

A table can be stored in different ways. Two matter here. A *row store* keeps each record's fields together, so one record occupies a contiguous stretch of storage. A *column store* keeps each column together instead. All the values of one column then sit side by side. The choice follows from the query shape of Section 2.1.2. A query that reads many rows but few columns is well served by a column store, which reads only the columns it needs. A row store has to read whole records and discard the fields the query never asked for [1].

The column layout is not better for everything. On disk, a row store holds each record in one place, while a column store spreads it across one place per column. A query that wants a single whole record therefore does less work on a row store. A query that reads many records does less work on a column store. It reads a large part of each column at once, instead of going back to that column for every record [1]. Analytical queries read many records, so they sit at the column store's end.

Writing works the other way around. One row's fields live in as many files as the table has columns. Changing a single row therefore means writing to all of them, while a row store writes once [1]. Many small writes are the transactional shape of Section 2.1.2, and a row layout handles them better.

The layout also decides how well the data compresses. Values in one column tend to resemble each other, while the fields of one record do not [1]. A column of country names holds many repeats of a few names. The record a name sits in holds a name, a date, and an amount, which have nothing in common. Compression algorithms perform better on data with low information entropy, so a column layout gives them more to work with [1].

DuckDB stores a table in row groups, which are horizontal partitions of the data [16]. Its default row group size is 122,880 rows [17]. DuckDB bases compression on row groups, and dictionary encoding is one of the schemes its storage format supports [16]. A dictionary from storage therefore describes one row group rather than the whole column. This is not

2. BACKGROUND

particular to DuckDB. Boncz et al. note that many systems apply dictionary compression to units smaller than the whole relation, such as row groups or data blocks [8]. This scope matters to the operators that read the column later. The layout also changes how a query can be executed, and not only how it is stored.

2.3 The execution engine

2.3.1 Compiled and vectorized execution

An engine has to turn a query plan into work on data. The classic approach moves one tuple at a time through the operator tree. Each operator asks its child for the next tuple, does its work, and passes the result on [1]. The design is simple, and every tuple pays for it. The interpreter makes a function call per tuple, and an analytical query has millions of tuples.

One way to avoid that call is to compile the query into machine code, which removes the interpreter altogether. Umbra does this, translating a query plan into machine code, as its predecessor HyPer did before it [40]. Both come from the group at TU Munich. CedarDB does the same, generating custom machine code for each query it executes [10]. Redshift also generates code, producing C++ for each query fragment and compiling it before execution [5]. Spark SQL generates code as well. Databricks replaced it in its own runtime with Photon, a vectorized engine written in C++ [6]. The other way is to process values in batches, so the interpretation cost is paid once per batch rather than once per tuple. DuckDB takes this route. A vectorized interpreted engine was chosen over just-in-time compilation “for portability reasons” [45]. Firebolt made the same choice when it built its runtime [41].

ClickHouse describes its engine as vectorized with optional code compilation, applied to an expression once that expression has recurred often enough [49]. Redshift went the other way and put a precompiled, SIMD-vectorized scan layer beneath its generated code [5]. Both approaches are in production use, and engines mix them, so vectorization is one established choice rather than the standard one. It is the one this thesis works within.

2.3.2 Vectors and data chunks

That second approach is called vectorized execution, introduced in MonetDB/X100 [9, 62]. An operator receives a batch of values, computes over the whole batch in a tight loop, and passes a batch on. The number of interpreter calls then falls by a factor equal to the size

of the batch. On a computationally intensive query such as TPC-H Q1, that can improve performance by two orders of magnitude [1]. Each loop also runs over a contiguous array of one type, which compilers optimize well and often turn into SIMD instructions [1].

Batching can be taken too far. MonetDB materialized every intermediate in full, which left it bound by memory bandwidth rather than by computation [9, 1]. The batch is therefore sized to stay in cache. Boncz et al. measured that any size between 128 and 8192 values worked well, and that performance fell away on either side of that band [9]. DuckDB calls such a batch a vector. It holds data of a single type at a default size of 2,048 values [18]. A data chunk is a horizontal subset of a table or an intermediate result [45]. It holds one vector per column, all of equal length, and serves as the unit operators pass between them.

2.3.3 Pipelines, sources and sinks

Operators are joined into pipelines. A pipeline begins at a source that produces chunks, such as a table scan, passes them through a chain of operators, and ends at a sink that consumes them. DuckDB uses a push-based model, in which chunks are pushed through the operator tree [19]. It has not always worked this way. The 2019 design pulled instead, with each node asking its children for the next chunk [45]. DuckDB switched to pushing in 2021 [42, 44]. Pipeline parallelism was one of the reasons. Pulling from the root node does not suit a design where pipelines are executed separately [42]. A sink is where a pipeline ends. It consumes its whole input and builds state from it before anything downstream sees a result.

Figure 2.2 shows a query with a join and an aggregation split this way. The scan on the join’s build side is a source, and the join itself is that pipeline’s sink. The same join appears again in the second pipeline, this time as an operator the probe rows pass through, and the aggregation closes that pipeline. One operator can therefore hold a different role in each pipeline it belongs to.

2. BACKGROUND

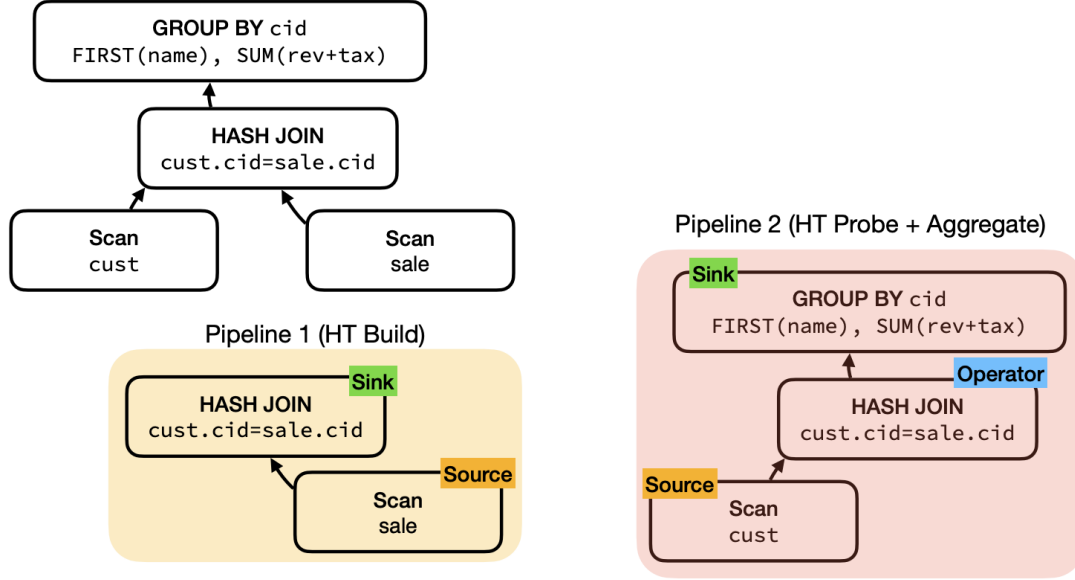


Figure 2.2: A query with a join and an aggregation, split into two pipelines. Image borrowed from [43].

2.3.4 Logical and physical vector formats

A vector stands for one value per row of the batch. What it actually stores can be much smaller. DuckDB’s vector formats “allow the system to store the same logical data with a different physical representation” [18]. A flat vector is a contiguous array with one entry per row, which is the uncompressed case. Other formats store a smaller set of values and describe the rest by reference.

Supporting every format in every operator does not scale. DuckDB describes the problem as a combinatorial explosion that would leave the engine with a giant code footprint [43]. An operator that must work on any representation can instead read a vector through a uniform view. DuckDB calls this the unified vector format and describes it as a generic view over the vector’s contents [18]. It supplies a data array, a selection vector, and a validity mask, so a row’s value is found by indexing the data array through the selection vector. Figure 2.3 shows the conversion for three of the formats. Nothing is decompressed and no value is copied. Velox describes its equivalent the same way, as collapsing the wrappings into at most one and handing back a base vector plus indices [54]. Reading through the view still costs something. An operator works once per row of the batch, even when the vector behind it holds only a few distinct values. The later chapters set out to avoid that per-row work.

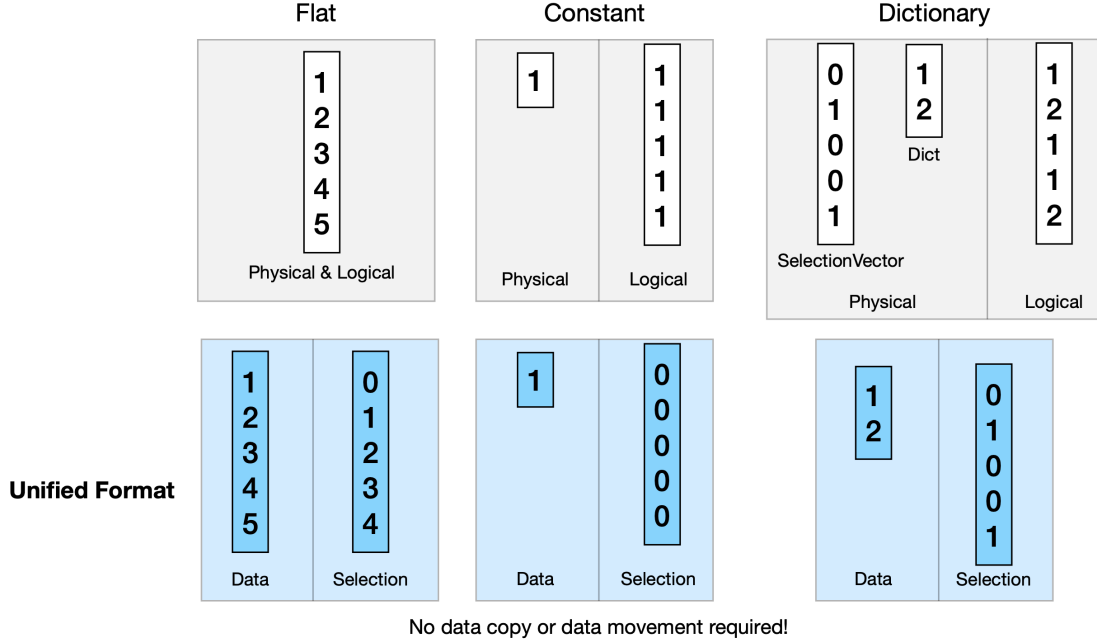


Figure 2.3: A flat, a constant and a dictionary vector, each converted to the unified format. Image borrowed from [43].

2.4 Compressed execution

A scan normally decompresses each column it reads, so every operator above it works on full values. Compressed execution is the alternative. The data stays encoded past the scan. Operators work on that form rather than decoding it first [38]. Graefe and Shapiro proposed leaving data compressed as long as possible, and decompressing only when unavoidable [29]. The structures a query builds while it runs then hold compressed values, so the query needs less memory [29]. It can also mean less work [38]. Chapter 3 covers the techniques that achieve both and what they are worth.

The technique has a cost in source code. An operator that works on compressed data has to know how it was compressed, so supporting several schemes multiplies the operator implementations a system needs [3]. This thesis works on dictionary encoding.

2.5 Dictionary encoding

Dictionary encoding is a lightweight compression scheme for columns that repeat a small set of values. The distinct values are stored once, in a structure called the dictionary. Each row then holds an index into that structure rather than a value [1]. A column of country

2. BACKGROUND

names with a few hundred distinct entries, spread over millions of rows, becomes a short list of names and a long list of small integers. The scheme pays when the values are large next to an index and the distinct values are few [29]. Boncz et al. state the limit plainly. Dictionary compression needs fully repeating strings, so it does nothing for strings that are similar but not equal [8].

The index is normally a fixed-width integer. A system can make every code the same width, which gives up a little compression and buys access patterns the processor handles well [1]. How wide follows from the data, since the bits needed for one value can be computed from the number of distinct values in the column. Abadi et al. kept their codes byte-aligned rather than bit-packed. A column store reads data efficiently enough that even modest compression makes a query limited by the processor rather than by reading. Once that happens, saving more space no longer helps, and unpacking bits would cost processor time [3].

The scheme is old. A 1993 survey of database compression lists it under the name abbreviations, a static mapping from values to short codes held in a lookup table [48].

What the encoding buys during execution is a change in the unit of work. An operator that understands it can do a piece of work once for each distinct value, then reuse the result for every row that shares that value. Łuszczak describes storing such results in an array indexed by dictionary code, so an expensive function is evaluated once per entry rather than once per row [38]. The work then scales with the number of distinct values rather than the number of rows.

Current systems carry the encoding in their own ways. Velox uses it throughout its execution engine, and says it does so to avoid copying data. A dictionary vector there is a base vector, a buffer of indices, and a nulls buffer, and one vector may carry several layers of dictionaries over a single base [54]. ClickHouse exposes it to the user instead, as a column type called `LowCardinality`, which it describes as changing both how data is stored and how it is processed [12].

DuckDB carries it as one of the vector formats of Section 2.3.4. A dictionary vector is stored as a child vector holding the values, together with a selection vector of indexes into that child [18]. The rows of a batch therefore reference a small set of shared values, and no value is duplicated across rows. Figure 2.4 shows a country column in both forms. Systems differ in these details, so the representation is a design choice rather than a fixed form.

Storage is where a dictionary vector ordinarily comes from. DuckDB emits one when it reads a dictionary-compressed column segment, and keeps the column compressed during

2.6 Dictionaries across consecutive chunks

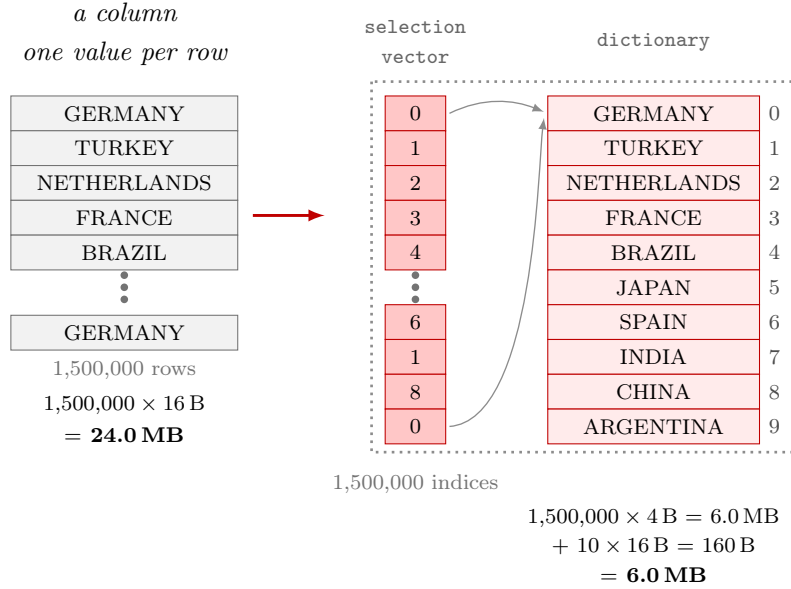


Figure 2.4: Dictionary encoding of a country column. Each row holds a four-byte index into a dictionary of ten names rather than a sixteen-byte string slot of its own. Two of the rows shown read the same entry.

execution rather than decoding it at the scan [18]. Whether an operator can rely on such a dictionary is a separate question.

2.6 Dictionaries across consecutive chunks

A dictionary can cover part of a column rather than all of it. Prior work treats that scope as the property which decides what an operator may do with the codes [38]. An operator sees one chunk at a time, so the question here is how many consecutive chunks one dictionary covers. A dictionary from storage covers one row group, which Section 2.2 put at 122,880 rows. That size is required to be a whole number of vectors [17], so at the default vector size of 2,048 [18] a dictionary spans sixty chunks. The sixty-first chunk may carry a different dictionary. An operator that has seen only the first sixty cannot tell.

Engines therefore label a dictionary, so that a consumer can tell whether two chunks share one. A DuckDB dictionary vector carries an identifier alongside its child vector and its selection vector [20]. Two chunks under the same identifier are backed by the same dictionary.

2. BACKGROUND

2.7 The hash join

A join combines rows from two inputs whose join keys match. Codd defined it as an operation over relations [13]. It says what the result is, not how to compute it. Choosing how is a separate step. DuckDB makes that choice in its physical planner, which selects an implementation for each logical operator [45]. For a comparison join it takes the hash join whenever the condition holds at least one equality, and a merge or nested-loop variant otherwise. This thesis modifies the hash join.

A hash join runs in two phases. In the build phase the engine reads one input in full and loads its rows into a hash table keyed by the join key. In the probe phase it streams the other input through that table, looks up each row's key, and emits an output row for every match. DuckDB builds on the right-hand input and probes with the left, and this chapter calls them the build side and the probe side.

The two phases are not symmetric, and that asymmetry is the point of the algorithm. The build side is read once to construct the table. Each probe row then costs about one hash and one lookup. The build side is usually the smaller of the two. In a star schema, one or more dimension tables act as the build tables, and a very large fact table is the probe [36].

The build side does not have to fit in memory. DuckDB splits it into partitions and loads as many as memory allows into the hash table. If they all fit, the join runs in memory from start to finish. If they do not, the rest is spilled to storage and joined in later rounds [33].

2.7.1 The row store and the pointer table

DuckDB's hash table is two structures working together. One is a row store, which holds the build-side rows. The other is an array of buckets, each pointing at a row in that store. That array is the hash table in the narrow sense. This chapter calls it the pointer table, and keeps *hash table* for the pair.

Each build row is written into the row store as one contiguous record. It holds the join keys and the payload columns, which are the build-side columns the query needs in its output. A layout descriptor fixes the order of the fields and the offset of each, so a field is reached by arithmetic rather than by search. Every record also ends with a single field of eight bytes. It holds the record's hash while the build runs, and *finalize* replaces it with a next pointer that links records together. The two uses never overlap in time, so they share the same bytes.

2.7 The hash join

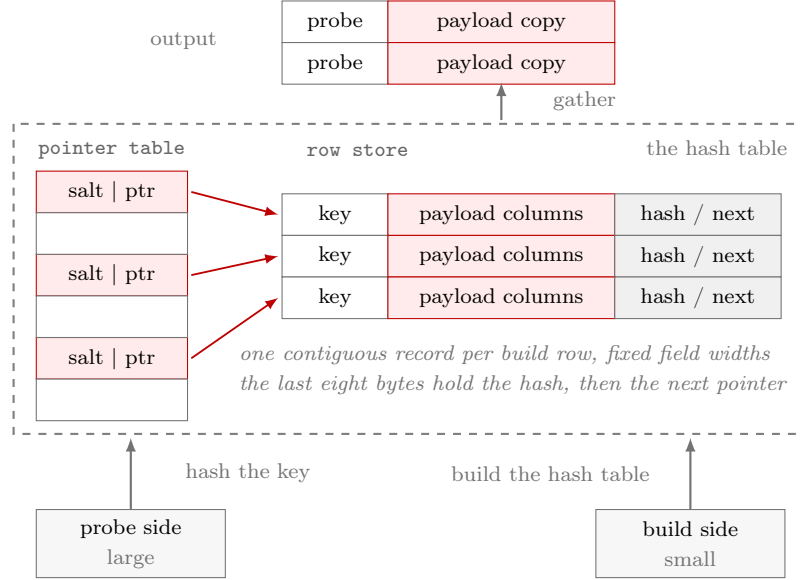


Figure 2.5: The two structures of the hash join and the phases that fill and read them. The build side is materialized into the row store, then finalize fills the pointer table, whose buckets carry a salt beside the pointer. A probe hashes its key to a bucket, and every match copies the payload columns into the output.

Field widths are fixed, which matters most for strings. A `VARCHAR` reserves a sixteen-byte slot in the record whatever the string’s length. DuckDB keeps the value inside that slot when it is twelve bytes or fewer, and otherwise keeps a pointer [21]. The bytes of a longer string sit in a heap block beside the row store, and the slot points into it. The reason for the split is in Umbra’s design, which uses the same sixteen-byte header. Inlining a short string avoids a pointer indirection. For a longer string the header keeps the first four characters beside the pointer, so some comparisons finish without following it [40]. A long string therefore costs a copy of its bytes into the join’s own heap, on top of the sixteen-byte slot. Chapter 5 builds on that distinction.

The pointer table is an array of 64-bit buckets, each holding a pointer to a record. A 64-bit pointer uses only its lower 48 bits for the address, so the top 16 are free [33]. They carry a salt taken from the record’s hash [33]. A probe that will find no match can be recognized from the salt alone, without touching the record [30]. The table is sized generously, at roughly two buckets per record [30].

Two records can want the same bucket for two different reasons, and DuckDB treats them differently. When the keys differ, the *collision* is resolved by linear probing and the newcomer takes another bucket. When the keys are equal, the records are linked into a chain. A chain therefore holds records of one key, and only one. The design is called

2. BACKGROUND

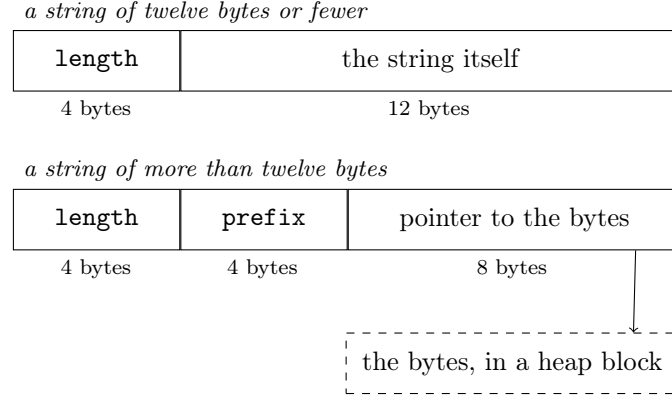


Figure 2.6: The sixteen-byte string slot in a build record, in its two forms. Only the second costs a heap copy. The same layout appears in Umbra’s string header [40].

Linear-Chained, and DuckDB adopted it in version 1.1.0 [30].

That arrangement moves work off the probe. The engine also learns during the build whether any chain formed. When none did, the probe skips chain following altogether.

2.7.2 The build phase

The build side is consumed before any probing begins, since a key cannot be looked up in a half-built table. This is the right-hand path of Figure 2.5. Threads take *morsels* of the input until it has all been read. Each thread hashes the join keys and materializes its rows straight into partitions, on a page layout that can be spilled. Partitioning and materializing happen together, so a tuple is never copied twice. When every thread has finished, the thread-local data is exchanged into one global state [33].

Partitions are chosen by radix, from a few of the middle bits of the hash. The lower bits give a record its bucket and the upper bits its salt, so the three uses must not overlap. The build starts with four radix bits, which makes sixteen partitions, and repartitions with more bits if a single partition will not fit in memory [33].

The pointer table is built once the build side has been read. DuckDB calls this step *finalize*. It runs one task per partition, so it is parallel as well. Every record is inserted into the table, and probing starts only after that finishes [33]. Finalize is the first moment at which the whole build side is known and sitting together.

Materialized pages are unpinned once written, so the buffer manager can evict them when memory runs short. Eviction has a consequence worth knowing. A page reloaded from disk may land at a different address, which invalidates every pointer into it. DuckDB keeps each page’s last known address beside the data, compares it on the next access, and

recomputes the pointers by offset when the page has moved [33]. A record pointer taken before a spill therefore cannot be trusted afterwards.

2.7.3 The probe phase

Once the table is built, the engine streams the probe side through it one chunk at a time, entering at the pointer table of Figure 2.5. For each chunk it hashes the keys. The lower bits of a hash give the bucket to start from, as Section 2.7.2 described.

A probe first compares the salt held in that bucket. A key that will find no match can often be rejected on the salt alone, without following the pointer to the record [30]. When the salt does not match, the probe may still have to try the next bucket, since a different key may have been displaced there by linear probing. The build recorded whether any collision happened on that bucket, in the uppermost bit of the salt, so the probe knows whether that is worth doing [30]. When the salt does match, the key itself is compared, because two different keys can share a salt.

That comparison is what locates the right bucket. It is not what filters the chain. Every record in a chain shares one key, so once the probe has found its bucket, every record hanging off it is a match. After the first hit the rest come off the chain with no further key comparison [30]. A probe row whose key occurs several times on the build side therefore produces one output row per matching record.

Each output row pairs the probe row's own columns with the payload columns of the build record it matched. Assembling those payload columns is where the cost of this join lies.

2.7.4 Producing the output

For every match the join has to place the build record's payload columns into the output chunk. It does this by *gathering*, the arrow marked gather in Figure 2.5. It reads each payload field out of the record in the row store and writes it into the output vector, one value at a time.

A small build side means the same few records match over and over, so the same payload values are read and written once for every probe row that matches them. A build column holding a few hundred distinct values is written into the output tens of millions of times, and the value has not changed between one copy and the next. The cost grows with the number of payload columns, and with the width of the values in them.

2. BACKGROUND

Copying of this kind is not a small cost. Graefe and Shapiro observed that copying accounts for a significant share of a database system’s processor and bus activity, and that materializing join output is faster when the records are shorter and less copying is required [29]. What the join produces here is a small set of values repeated across many rows, which is what Section 2.5 described a dictionary as being for.

2.8 Dictionary-aware operators in DuckDB today

Some of DuckDB’s operators are already dictionary-aware. Two are closely related to the work of this thesis and worth examining before the contributions.

The idea behind the first is older than DuckDB. In a star schema a dimension key is often a sorted, contiguous list of identifiers, so a foreign key value is already the position of the row it refers to. Abadi’s invisible join exploits that. Once the qualifying fact-table rows are known, the dimension columns the query needs are extracted by position, which is a plain array lookup rather than a hash probe [2].

DuckDB applies the observation to the lookup itself, in what it calls the *perfect hash join*. When the build-side key is an integer from a small, bounded range, the position of a key is the key minus the smallest key in that range. There is no hash function, no salt, no collision handling and no chain, and a lookup is one subtraction and one array access. The conditions are strict, and the engine checks them once the build side is known. The join must be an inner join on a single integer equality, the key range must be small enough to allocate as an array, and there must be no duplicate keys. If any of them fails, the engine falls back to the general hash join. The perfect hash join stores each build-side output column as one array indexed by key. It can therefore emit those columns as dictionary vectors pointing into the array, rather than copying values into every output row. That is a by-product of the array layout, not an attempt to compress the output.

The second is the hash aggregate’s handling of a dictionary-encoded grouping key. When the key of a `GROUP BY` arrives as a dictionary vector from storage, the aggregate does not look up every row. It resolves each distinct value once, records where that value maps to in its table, and reuses the result for every row that shares the value [22]. A batch of thousands of rows with only a few distinct keys then needs only a few lookups. What it resolved is kept across batches for as long as the dictionary stays the same [22], which is the scope question of Section 2.6.

2.8 Dictionary-aware operators in DuckDB today

The two use dictionaries in opposite directions. The perfect hash join is a producer, passing a dictionary to the operator downstream. The hash aggregate is a consumer, using one that arrives on its input.

2. BACKGROUND

3

Literature Study

This chapter surveys the work this thesis stands on and marks where it departs. Section 3.1 traces how the field arrived at compressed execution, what the technique is worth, and what it costs in source code. Section 3.2 turns to the scopes a dictionary can have, what each one buys, and why none of them serves an operator that materializes its input. Section 3.3 sets out what an operator can do with a dictionary it is handed, and what each technique needs from that dictionary’s scope. Section 3.4 gathers the approaches the field has taken to compressing the structure a hash join builds. Section 3.5 draws the four together.

3.1 Compressed execution

Compressed execution is query processing in which the operators work on compressed data rather than on decompressed values. It can lower the memory a query needs and make the query faster. This section covers how the field arrived at compressed execution, what it is worth, and the challenges it brings.

3.1.1 From storage to execution

Compression entered database systems as a way to store less on disk. Graefe and Shapiro proposed keeping data compressed far longer than that, all the way into query execution, so that the operators themselves work on compressed values [29]. Their reason was about memory. Compression makes the data smaller, so a fixed amount of memory holds more of it. That matters most to a join. A hash join loads its build side into memory, then looks up every probe-side row in it. When the build side is too large, the excess is written to disk and processed in a later pass. Compressing the build side means more of it stays in memory, so less of it goes to disk [29].

3. LITERATURE STUDY

Columnar database systems made compressed execution a design goal. C-Store’s designers wrote that the executor should work on the compressed representation until values have to be handed to an application [51]. Their plans contain an explicit Decompress operator, so losing the compressed form is a step that somebody chose and that a reader of the plan can see [51]. Most engines have no such step. A column loses its compressed form wherever an operator declines to read it, and nothing in the plan shows that it happened.

3.1.2 Direct operation on compressed data

How long a value stays compressed is a choice, and four points along that scale have names. *Eager decompression* decompresses a column as soon as it is read [3]. *Lazy decompression* waits, and decompresses a column only at the first operator that needs its values [3]. With eager and lazy decompression, a column that has been decompressed stays decompressed for the rest of the plan, since nothing compresses it again during execution [11].

Transient decompression is different. Some operators cannot do their work on the compressed form, and a sort is the clear case, since it has to compare the real values to order them. Such an operator decompresses the values it needs for its own use, does its work, and forwards the column exactly as it arrived, still compressed [11]. The decompression never leaves the operator. *Direct operation* is the end of the scale, where an operator works on the compressed form and never decompresses it at all [3].

Abadi et al. measured direct operation against eager decompression on aggregation queries, one scheme at a time, over data with 1000-record sorted runs. Run-length encoding ran 3.3 times faster and bit-vector encoding 10.3 times faster. Dictionary encoding ran 3.94 times faster when the group-by and the aggregate were on the same column, and 1.1 times faster otherwise [3].

3.1.3 The cost of compression-aware operators

Working on compressed data has a cost in source code. An operator that reads a compressed column has to know how that column was compressed. A system with several compression schemes then needs a version of each operator for each one. A join reads two inputs at once, so it needs a version for every pair [3]. A scheme is chosen per column, so an operator over a tuple of several columns needs a version for every combination across them [34]. A typical operator then turns into a huge set of if statements, one for each of the compression combinations it might be handed [1, 34].

SQL Server’s column store scan reached over ten thousand specialized implementations, compiled ahead of time for every combination of data type, encoding parameter and filter type, and selected through a function pointer [35]. That count is for the scan alone.

A possible answer is to hide the compression scheme behind a small interface. Abadi et al. let an operator question a block rather than ask which scheme produced it. The operator asks whether the block holds a single value, whether its values are sorted, and whether its positions are contiguous, then takes its shortcut from the answers [3]. The choice is made at runtime, from the data, which is a form of adaptive execution.

A compiling engine cannot afford that dispatch. In HyPer, where each block may carry its own layout, the code paths a scan needs grow as p^n for n attributes stored p ways. Two attributes over six representations already need thirty-six, so the cost lands on compilation time. Lang et al. therefore feed compiled query pipelines from an interpreted vectorized scan, which stays precompiled and can express adaptive algorithms [34]. Redshift reached the same arrangement from the other side. Inlining decompression code for every column made compilation slow on wide tables, so a precompiled vectorized scan layer now sits underneath code that is still generated per query [5].

HyPer evaluates its predicates on the compressed representation, then unpacks the matches for the pipeline above [34]. Redshift’s scan layer writes the qualifying column values to arrays that the downstream steps read [5]. Neither lets a compressed value past the scan, so for every operator above it the effect is eager decompression.

3.2 Dictionary scope

Ghiță et al. use the name black-box compression for the schemes an engine implements in code, such as run-length encoding and bit-packing. The decompression logic is fixed there, so query operators cannot work on the compressed form without running that code [26].

A dictionary that reaches an operator during execution is different. It arrives as a small set of distinct values together with one index per row, both laid out as plain arrays. Nothing has to be decompressed to reach the indices, so an operator could work on them rather than on the values. Hiddenness is therefore not what keeps operators away from a dictionary. What an operator cannot see is the dictionary’s scope. Scope is how much data one dictionary describes, and it decides whether working on the indices is safe. This section covers the scopes the field has named, what each one costs, and why none of them serves a materializing operator, one that builds a structure in memory and copies values into it.

3. LITERATURE STUDY

3.2.1 The scopes prior work names

Foufoulas et al. treat the choice of scope as the research problem rather than as a detail of the format. A per-block dictionary covers one block. A global dictionary covers a whole table column. Their account of the trade between them runs in both directions [24]. When a query filters a dictionary-encoded column, the scan does two things. It looks the value up in the dictionary to find its index. It then reads the stored index of every row and compares it against that one. The two scopes divide that work differently. A per-block dictionary has to be looked up once in every block. In exchange, a block whose dictionary does not hold the value needs none of its stored indices read. A single global dictionary is looked up once for the whole column, and the index it returns is valid in every block. Every block's stored indices are therefore read, unless the block can be skipped for some other reason [24]. Foufoulas et al. measured both parts on one filtered scan. Per-block dictionaries spent 270 milliseconds on dictionary lookups and 7.4 on reading stored indices. A single global dictionary spent 0.036 milliseconds on lookups and 62 on reading stored indices [24]. Each scope is cheap where the other is expensive. Both schemes here are their own C++ implementations, written for the comparison, and the authors call the evaluation preliminary.

Łuszczak names three scopes rather than two, and works out what each operation requires. A partition-wide dictionary covers one partition of a column, so its indices mean something only inside that partition. A column-wide dictionary covers a whole column. Most of the literature calls that a global dictionary, and this chapter uses the shorter name. A domain-wide dictionary covers every column drawn from the same domain, so two different columns carry indices that mean the same thing. Comparing a column against a constant needs column-wide scope. A join needs domain-wide scope, since the encoding on both sides of the join has to be identical [38].

3.2.2 Global dictionaries

Three commercial systems use a global dictionary. SAP HANA compresses every column of its main store with a sorted dictionary, mapping each value to an integer identifier and bit-packing those identifiers [23, 50]. DB2 with BLU Acceleration partitions each column's values by frequency and builds a dictionary for each partition, from a sample taken when the table is first populated, then handles values the sample missed with page-level dictionaries [46].

SQL Server keeps two dictionaries at once. A column segment may reference a shared global dictionary together with a per-block dictionary for a single row group. The global dictionary holds the most common values, and the per-block one holds what the global dictionary missed [35].

Gubner et al. give four reasons why a global dictionary is not widely adopted. It has to reside fully in memory. Parallel or distributed loading would need continuous synchronization to keep it consistent. It amounts to a mandatory secondary index on every string column. Inserts, updates and deletes force re-coding or garbage collection [31].

Łuszczak reaches the same conclusion independently, naming concurrency across parallel transactions and the re-encoding forced when indices outgrow their width [38]. She adds one the others do not, that a column whose values cluster locally compresses better under several small dictionaries than under one [38].

The width of an index is where that re-encoding comes from. Every stored index is a fixed-width integer, and the width comes from how many values the dictionary holds. A dictionary of 256 values needs one byte per index, and one more value pushes every index to two. Zukowski et al. name the consequence, that widening a shared dictionary by one bit triggers recompression of everything already stored [61].

Each of the three systems above pays a specific price to hold the wide scope. BLU freezes its dictionary at load time and patches the remainder per page [46]. SQL Server bounds its global dictionary and keeps a per-block one beside it [35]. HANA carries a second dictionary in a delta store, held in a cache-sensitive B+-tree and merged into the main storage periodically [23]. All three prices are paid for the same reason. The dictionary covers a whole column of stored data, and that data changes.

HANA's arrangement shows what the wide scope costs in practice, and Figure 3.1 follows a record through it. It arrives in a write-optimized structure that applies no compression, then moves into a column-format delta. That delta's dictionary is left unsorted for speed, so point queries need a secondary index over it [50]. Only the merge into the main store gives the column the sorted, bit-packed dictionary described above.

Trinh recently built a global dictionary for Parquet inside Delta Lake. Parquet normally gives each column chunk its own dictionary. He replaced those with one dictionary per column, held in a separate file beside the data and found through Delta Lake's transaction log. Measured against Parquet's own per-chunk dictionaries, the files shrank by 11 percent on TPC-DS and grew on all three real datasets [53]. Parquet's fallback rule is why. One value missing from the global dictionary sends its whole column chunk to plain encoding, so it has to cover everything to save anything. Splitting the index space, a shared range

3. LITERATURE STUDY

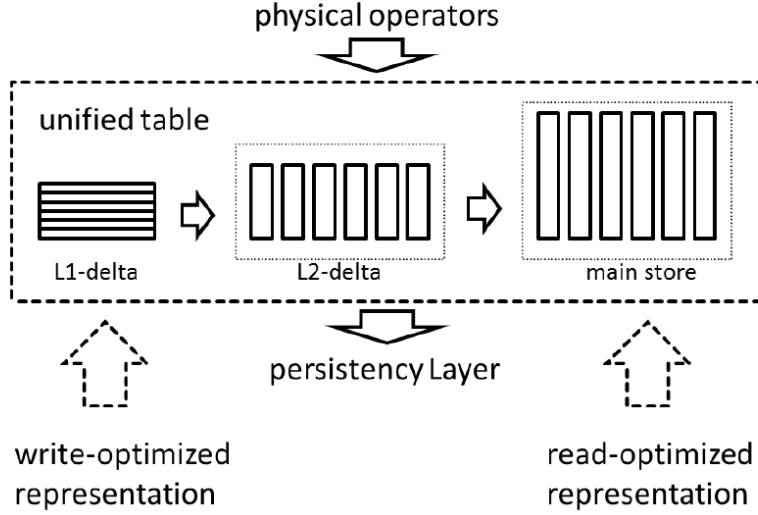


Figure 3.1: The life cycle of a record in SAP HANA. It arrives in the write-optimized L1-delta, which applies no compression, passes through the L2-delta, whose dictionary is left unsorted, and is merged last into the read-optimized main store. Image borrowed from [50].

for the global dictionary and a private range for whatever a chunk adds, recovered the loss on all three [53].

The Informix Warehouse Accelerator answers the same problem with a place for the leftovers. It partitions each column’s dictionary and stores rows in cells formed from those partitions. A row holding even one value it cannot encode goes instead to a single catch-all cell, which keeps the whole row unencoded even where its other columns could have been encoded [36]. Figure 3.2 shows both.

3.2.3 Per-block dictionaries

Per-block scope avoids recompressing a whole column. A block that is dictionary encoded gets its own dictionary and its own index width, so a new value in one block never touches the other blocks, already written [61].

A per-block dictionary is also smaller. A column’s values are often distributed differently in one part of it than in another, so a block holds only some of the column’s distinct values [1]. A dictionary built for that block has fewer entries than a global one, and its indices are narrower as a result [24].

The price is redundancy. Lang et al. chose per-block dictionaries for Data Blocks and named the cost, that identical strings appearing in several blocks have to be stored in each of those blocks’ dictionaries [34]. A value spanning many blocks is therefore stored

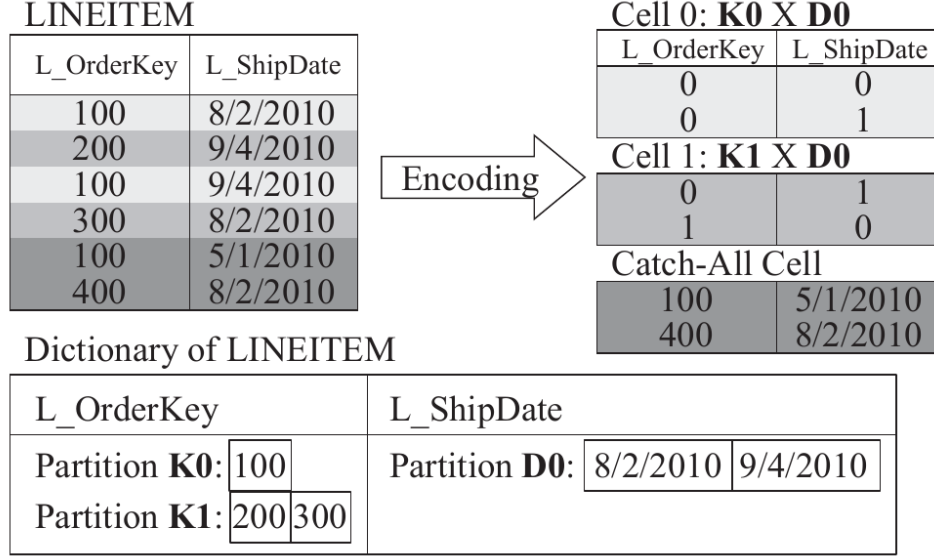


Figure 3.2: Encoding with a catch-all cell in the Informix Warehouse Accelerator. Two dictionary partitions of L_OrderKey and one of L_ShipDate give two encoded cells. The two rows carrying a value outside those partitions are kept whole and unencoded. Image borrowed from [36].

once in each of them, so the saving shrinks as the scope narrows. They also name what amortizes it. Each column of each block takes whichever scheme suits it, so a column held as a dictionary in one block may be truncated or reduced to a single value in the next [34]. Figure 3.3 shows both.

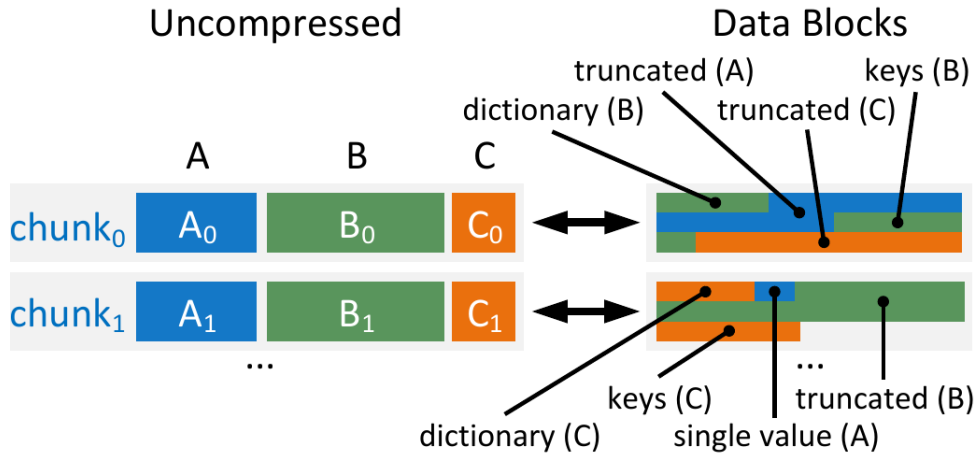


Figure 3.3: Data Blocks choose a compression scheme for each column of each block. Column B is dictionary encoded in the first chunk and truncated in the second, while column C is the other way round. Image borrowed from [34].

3. LITERATURE STUDY

Boncz et al. state the general limit, that many systems apply dictionary compression to units smaller than the whole relation, such as row groups or data blocks, and that this may limit its effectiveness [8].

Formats also need a rule for when to stop building a dictionary. Parquet uses a size cap of one megabyte, while ORC uses the ratio of distinct values to rows. Zeng et al. prefer the ratio, since tuning it does not depend on the row group size [59].

Zukowski et al.’s PDICT lets a block reuse the dictionary of a previous block [61]. A dictionary can therefore outlive the block that built it. Nothing in the scheme records that it has, so a consumer cannot tell reuse from a fresh dictionary and has to treat every block as new.

Differential dictionaries. Foufoulas et al. go further and build a scope between the two. One page carrying its own dictionary is followed by a run of pages whose dictionaries hold only the values not already seen in that run. A cost function decides per page whether to start a fresh dictionary or continue the run [24]. The property that matters here is not the compression ratio. New values are appended and the indices assigned earlier never change meaning, so a consumer holding a resolution from the first page of a sequence still holds a correct one on the last [24].

3.2.4 The cost to materializing operators

Neither scope helps most operators where the values are copied. In some systems, Gubner et al. observe, pushed-down selections are evaluated on the dictionary while every other operation decompresses, because the dictionary is only available to the table scan operators [31]. A hash join or a group-by therefore typically allocates heap memory for every string it holds, which they call very inefficient and a sparsely researched topic [31]. One structure reaches across a whole query. Gubner et al.’s Unique Strings Self-aligned Region, or USSR, is built afresh for each query by inserting frequently occurring strings as the query runs [31]. It is fixed at 768 kilobytes so that it stays in the processor cache, and holds at most 32 thousand strings [31]. Insertion into it can fail, so no operator may assume that a given string is there [31]. It is a cache with query-wide reach rather than a scope, since it describes no unit of data completely. That is enough to make an operation faster and not enough to let an operator commit to a layout in advance.

Afrozeh brought the same idea into DuckDB as the Unified String Dictionary [4]. One region per query holds each distinct string once. An operator that would otherwise copy a long string into its own heap points into the region instead, so the bytes are stored once for the whole query rather than once for every row that holds them. Two operations get

cheaper as well. Each stored string carries a precomputed hash, so hashing becomes a load. Entries are unique, so two strings are equal exactly when they sit in the same place.

Three things limit it. Only strings too long to store inline qualify, since a shorter one already sits inside the row that holds it. The region has a fixed size, so a string that does not fit goes back on the heap and no operator can assume a string is there. DuckDB also has other optimizations that reach these columns first and leave the region nothing to save. On TPC-DS, IMDB and ClickBench it made no difference either way [4]. It is not currently part of DuckDB.¹

Every scope named in this section is a unit of stored data, whether a block, a run of pages, a partition, a column, or a domain. Each one is fixed before a query runs, by whatever wrote the data down. A dictionary that an operator builds while a query runs describes none of those units, so the axis as the field draws it has no point where such a dictionary sits.

3.3 Dictionary-aware operators

There is more than one way to exploit a dictionary during execution, and different operators take different routes. This section covers three of them, what each one saves, and what each one needs from the dictionary's scope.

3.3.1 Operating on indices instead of values

An operator can often do its whole job on the indices and never look at the values at all. Monet stored low-cardinality string columns as one or two byte integers [7]. A query filtering such a column for the string `MAIL` never compares that string against a row. The dictionary is searched once to learn that `MAIL` has index 3, and the filter then tests each row's stored index against 3 [7].

Zukowski et al. name two forms of this. A selection can run against the index, testing for the integer that stands for a value rather than for the value itself. An arithmetic operation can sometimes run over the dictionary alone and leave every stored index untouched, a technique they report under the name enumeration view [61].

A group-by has to keep one running result for each distinct value of the grouping column, and it normally finds that result by hashing the value and looking it up. Holloway et al.'s group-by does neither. A dictionary already gives each distinct value a small integer, so the operator allocates one result slot per dictionary entry and uses a row's stored index as

¹<https://github.com/duckdb/duckdb/pull/18184>

3. LITERATURE STUDY

the position of its slot. The grouping column is never decompressed [32]. They state the division generally. A field that is only compared or grouped on stays compressed, while a field being aggregated is decompressed [32].

SQL Server takes this furthest on the scan. An arbitrary filter on a dictionary-encoded column becomes a bitmap with one bit per dictionary entry, which pays because a dictionary usually holds far fewer entries than there are rows referencing it [35]. A string equality becomes a search for one integer in the encoded column. They measured the new scan against the one it replaced, on TPC-H at 300 gigabytes with the data cached. A filter on a numeric column fell from 664 milliseconds to 68, a factor of 9.76, and a string equality fell from 694 to 147, a factor of 4.72 [35]. That speedup depends on the dictionary’s scope. SQL Server drops the bitmap when the dictionary is larger than the row group [35].

SAP HANA reports the same at the scan and the aggregate, running SIMD algorithms directly on the compressed data and exploiting run-length encoding to compute aggregates [23].

3.3.2 Reusing a resolution across batches

Resolving each distinct value once and reusing the answer is a named technique. Łuszczak calls it results caching. The result of an expensive function is stored in an array addressed by the index and read from there afterward, so the function runs once per dictionary entry rather than once per row [38]. She states what it needs, which is at least partition-wide scope. Entries need not be unique and the encoding need not preserve order [38].

DuckDB does this in its grouped aggregate. When a grouping key arrives as a dictionary, the operator checks each entry for a lookup it has already made and reuses the stored answer where it finds one [22]. Whether that cache survives into the next batch is decided by the identifier the dictionary carries. The code says so in two comments, that a dictionary without an identifier cannot be cached across vectors and one with an identifier can [22].

Gubner et al. cache a different kind of answer. Every string in the USSR has its hash stored in the eight bytes before it, so hashing a string that is already there is a load rather than a computation [31]. Their entries are unique, so two strings in the structure are equal exactly when their pointers are equal [31]. The structure lasts for the whole query, so both answers are computed once however many batches a string appears in. On a string group-by they measured comparison 2 to 50 times faster and hashing 4 to 80 times faster [31].

3.3.3 Joining on encoded values

Graefe and Shapiro set the condition. Fixing a compression scheme per attribute lets a column be compared against a compressed constant, and fixing it per domain is stronger, since it lets compressed values that arrived from different sources be compared [29]. Keys and foreign keys share a domain, so a join on compressed keys returns what a join on decompressed ones would [29].

Not every column has to meet that condition. Łuszczak observes that attributes a join only copies can stay encoded, since nothing compares them against the other side [38]. She names one exception, that indices drawn from partition-wide dictionaries could be mixed together in the join’s result [38].

Lee et al. explain why a system does not simply encode a domain once. It cannot know in advance which columns a user will join on [36]. Even if it could, the hub of a schema is a primary key whose values are close to uniformly distributed, which forces fixed-length indices and gives up the skew that makes a per-column dictionary compress well [36]. The Informix Warehouse Accelerator gives each column a dictionary of its own, built at load time from the whole column, and reconciles the two sides of a join at run time instead, in a procedure they call encoding translation. It maps one join column’s indices into the other’s [36]. The direction is chosen for cost. Translating both sides into a fresh shared encoding would mean a decompression and a re-encoding on each side. They translate the build side into the probe side’s encoding instead, since the build side is usually the smaller of the two [36].

DB2 with BLU Acceleration makes the same choice. Join keys use the encoding of the outer, larger input, and the inner’s join columns are converted into that space during the scan of the inner, because inners generally have smaller cardinality than outers [46].

The same condition turns up under a scheme that is not a dictionary. FSST compresses a string by replacing frequently occurring substrings of one to eight bytes with a single byte each, drawn from a symbol table shared across a block of strings [8]. Equality can be evaluated on compressed values only while both operands share one symbol table [8]. In their prototype integration into Umbra, a join whose two string columns had been compressed under different symbol tables had to decompress before matching [8].

3.4 Compressing the hash table

This section covers the different approaches the field has taken to compressing a hash table, what each one narrows, and what each one derives its encoding from.

3. LITERATURE STUDY

3.4.1 The hash table as the largest structure

A hash join cannot probe until it has read its entire build side, so it holds every build row in memory at once, in the hash table it builds. Each row carries the join keys together with every build-side column the query needs in its output. The probe side streams past one batch at a time and is released as it goes, while the hash table stays until the last probe row has been matched.

That makes the hash table the structure most likely to set a query’s peak memory, which is why Gubner et al. compress the table itself [31].

How large that table will be is not known when the query is planned. Leis et al. compared more than 100,000 cardinality estimates against the truth and found misestimates by a factor of 1000 or more to be routine across every system they tested [37]. The errors run systematically toward underestimation once a query has several joins, so a build side is more often larger than planned than smaller [37]. Compression leaves the row count as unpredictable as it was. It reduces what each row costs, so the table is smaller whatever the count turns out to be.

3.4.2 Compressing what the table holds

The rows of a hash table are fixed width, including the payload columns, which are the build-side columns a query needs in its output. Compressing a hash table therefore means narrowing those fixed-width slots.

Vectorwise. Gubner et al. compress the rows of a hash table by three composable means. Rows are bit-packed against the smallest and largest value each column is known to hold, so a column whose values all fall in a narrow range needs only the bits that range requires. Each row is split into a hot area and a cold one. The third is the USSR, which lets a row hold a short slot number rather than a full pointer [31]. Bound-driven packing has a floor they state plainly. A string in a row is a 64-bit pointer into a heap, and on current hardware an address needs 48 of those bits, so packing from bounds cannot take a string column below 48 bits [31]. Their string structure is what gets past it, since a slot number in a region of at most 65,536 slots fits in 16 [31]. Figure 3.4 shows the structure. Their join probe micro-benchmark reached 2.5 times including tuple reconstruction, with the gain rising as the build side outgrew the cache [31].

The Informix Warehouse Accelerator. Lee et al. encode payload columns during the build itself, in what they call on-the-fly encoding [36]. On a customer business intelligence workload over 100 gigabytes of TPC-DS it improved every query, by 17 percent on

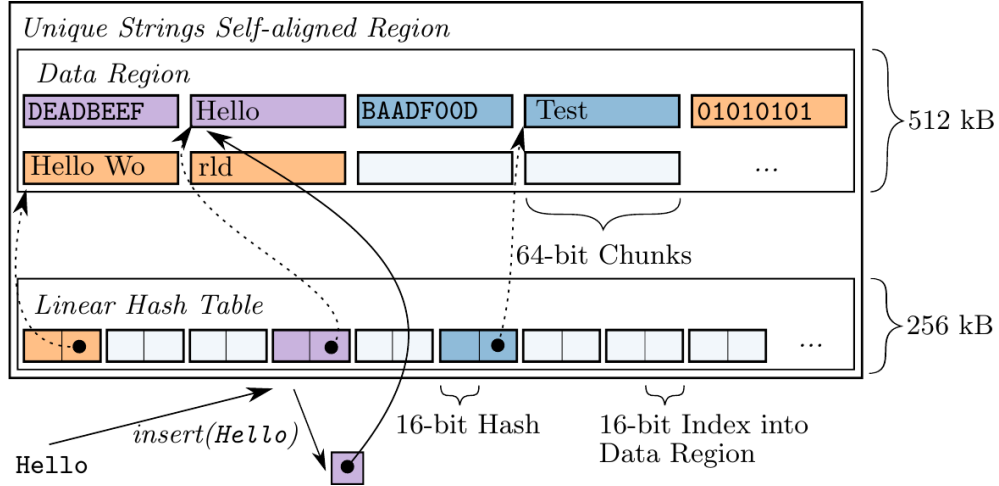


Figure 3.4: The Unique Strings Self-aligned Region. Each string sits in the data region with its hash in the bytes just before it. A bucket of the linear hash table holds a sixteen-bit hash extract beside a sixteen-bit slot number pointing into that region. Image borrowed from [31].

average and up to 52 [36]. As each build row arrives, the join looks its payload value up in a mapping table it is building as it goes. A value already seen reuses its index, and a new one is given the next free index. What the hash table receives is the index rather than the value [36]. The width of that index cannot be known while rows are still arriving, since the number of distinct values is settled only once the whole build side has been read. An index can never change after it has been issued, because rows already written hold it. They allocate a generous width during the build and compact the space afterwards, once the true count is known [36]. The indices are new ones rather than the column's own. A filter or a join predicate cuts a column's cardinality, so a dictionary built during the join is smaller than one built at load time from the whole column [36]. What a downstream consumer meets is therefore a space of indices that did not exist when the join started.

DB2 with BLU Acceleration. Raman et al. partition each column's values by frequency and build a dictionary for each partition, from a sample taken when the table is first populated, then handle values the sample missed with page-level dictionaries [46]. Joins run over encoded data for both keys and payloads, which is what they give as the reason hash tables stay compact enough to fit into low levels of on-chip cache [46]. A payload column is encoded with its own dictionary, the one built at load time. Values that were never encoded at load time are given a separate dictionary, constructed while the query runs [46]. A payload never has to agree with anything on the other side of the

3. LITERATURE STUDY

join, since it is carried rather than compared, so a column-wide dictionary is enough for it. The encoding also changes the table’s own layout. Their hash table uses an indirection bitmap to almost entirely avoid collisions while leaving no empty buckets, and where the encoded nature of the join keys allows they make it collision-free outright [46].

Compressed Materialization. DuckDB narrows the same rows from outside the operator. Kuiper et al.’s optimizer pass, shipped since version 0.9.0, inserts a projection that compresses a column before a materializing operator and another that decompresses it after [33]. Its authors state the reason plainly, that implementing compressed execution as an optimizer rather than inside hash tables centralizes the logic and makes it applicable to all operators without modification [33]. The projections apply to the columns going into the operator, and the authors note that the transformation is safe even for a join key, as long as the other side is transformed the same way [33]. An integer column is narrowed by frame of reference. The optimizer takes the smallest value the column is known to hold, subtracts it from every value, and casts the result to the narrowest integer type that still fits [33]. A column of years between 1990 and 2020 spans 31 values, so subtracting 1990 leaves a number that fits in one byte. A string column is narrowed only when its maximum length is known to be small. The characters are packed directly into an integer, with nothing looked up anywhere. A column of three-letter currency codes takes three bytes per value, which fit inside a 32-bit integer where the string representation would have taken sixteen, saving twelve bytes per value [33]. The bytes are reversed as they are packed, so comparing two of those integers gives the same answer as comparing the two strings [33]. Both routes narrow a type rather than a set of values. A wide column holding few distinct values is out of reach of either, since neither the range of an integer nor the maximum length of a string says how many distinct values there are.

SAHA. Zheng et al. take the opposite route with a hash table built for string keys, whose code they state has been merged into ClickHouse [60]. It keeps the strings themselves and is rebuilt around them, dispatching each key on its length into one of five sub-tables. Keys of two bytes or fewer go to an array lookup table that stores no key at all. Keys up to 24 bytes are reinterpreted as one, two or three eight-byte integers held directly in the slot. Anything longer keeps a pointer and a saved hash [60]. Figure 3.5 shows the arrangement. The threshold has two reasons. They observe string datasets in which 80 percent of values are shorter than 24 bytes, and using more integers would offset the gain through the extra copying [60]. The hash is saved only above that length, since recomputing it is expensive while comparing two long strings on every probe is worse [60]. Their saving is on short strings, and above 24 bytes they report no memory optimization at all, using the same space

as any table that saves a hash [60]. They name dictionary encoding as the alternative and give its limit, that it works only when a string column has low cardinality, since otherwise maintaining the dictionary becomes too expensive. They hold that the two compose rather than compete [60].

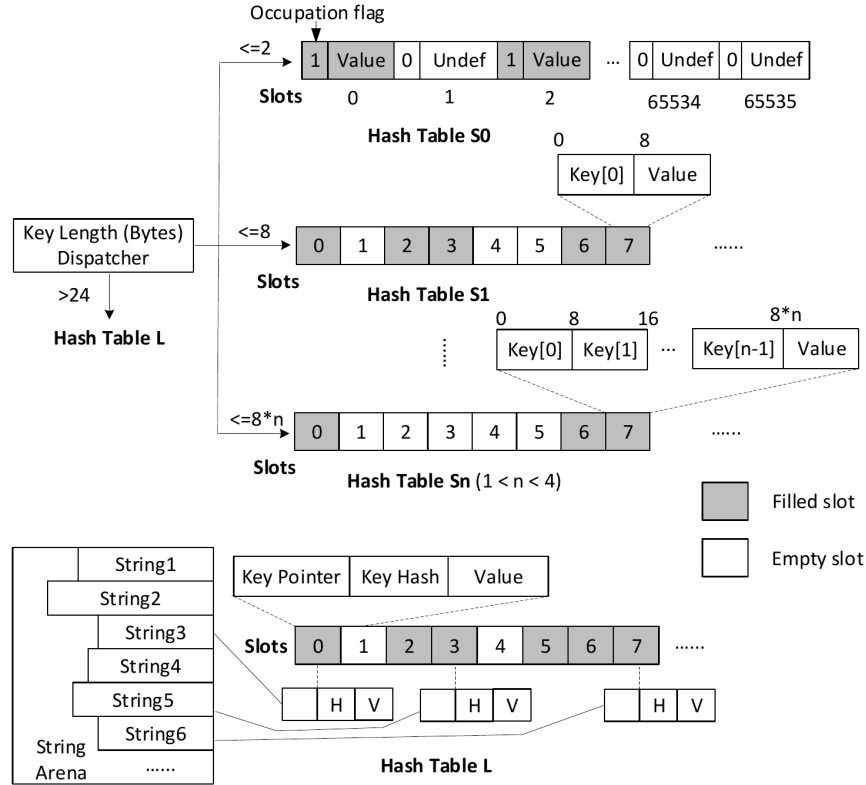


Figure 3.5: SAHA dispatches a string key on its length. Keys of two bytes or fewer reach an array lookup table that stores no key. Keys up to 24 bytes are held as one, two or three eight-byte integers in the slot itself. Longer keys keep a pointer into a string arena beside a saved hash. Image borrowed from [60].

3.5 Summary

This chapter has covered compressed execution, the scopes a dictionary can have, what an operator does with one, and what the field has done to compress the structure a hash join builds. Table 3.1 summarizes our survey by highlighting the benefit of each discussed topic and its adoption in widely used data management systems.

3. LITERATURE STUDY

Method	Benefit	Adoption status
Compressed execution	The operators work on the compressed form rather than decompressing it, which avoids the decompression cost and keeps intermediates small.	Shipped in DB2 with BLU Acceleration, SQL Server, SAP HANA and DuckDB. Supporting several encodings costs a version of every operator per scheme, which SQL Server met with over ten thousand specialized scan implementations.
Global dictionary	One dictionary for a whole column, so an index resolves the same way in every block.	Held by SAP HANA, DB2 with BLU Acceleration and SQL Server, each paying for it differently, by freezing the dictionary at load, by bounding it, or by carrying a delta store. Gubner et al. name four costs that keep it from wider use, chief among them that it must stay resident and that extending it forces re-coding.
Per-block dictionary	Narrow indices and small dictionaries, since a block holds only some of a column’s distinct values.	The common choice, in Parquet, ORC, Data Blocks and DuckDB. The same value is stored again in every block that holds it.
Differential dictionary	A dictionary spans a run of pages rather than one block. New values are appended and earlier indices never change meaning, so a resolution stays correct across the run.	Built deliberately by Fofoulas et al., and reached incidentally by PDICT, which lets a block reuse a previous block’s dictionary. PDICT records nothing, so a consumer has to treat every block as new.
Runtime dictionary	Built after filtering, so smaller than one built over the whole column at load time. It needs no agreement with storage.	The USSR is capacity-bounded and its insertion can fail, so no operator may assume a string is present. On-the-fly encoding mints a fresh index space during the build, so a consumer meets indices that did not exist when the join started.
Operating on indices	An operator does its whole work on the indices. A filter searches the dictionary once and then compares small integers, and a group-by uses a row’s index as the position of its result slot.	Done at the scan in Monet, SQL Server and SAP HANA, and extended to the group-by by Holloway et al. It applies to a column that is only compared or grouped on, since an aggregated column has to be decompressed. SQL Server abandons its dictionary bitmap when the dictionary is larger than the row group.

3.5 Summary

Method	Benefit	Adoption status
Reusing resolution a	A distinct value is resolved once and the answer is stored against its index, so an expensive lookup, hash or comparison runs once per dictionary entry rather than once per row.	Used in DuckDB’s grouped aggregate and by the USSR. Łuszczak states what it needs, which is at least partition-wide scope.
Joining on encoded values	The two sides of a join match on indices rather than values, so no key is decompressed during the probe. That needs domain-wide scope, since both sides have to agree on one encoding.	Done by the Informix Warehouse Accelerator and by DB2 with BLU Acceleration. Neither shares one encoding between the sides. Both reconcile them at run time, translating the smaller input’s indices into the larger input’s space. Graefe and Shapiro state the condition, which is a scheme fixed per domain rather than per column.
Compressed hash table	A smaller hash table can lower a query’s peak memory and keep more of the build side in memory and in cache.	Present in Vectorwise, the Informix Warehouse Accelerator, DB2 with BLU Acceleration and DuckDB. Most of them narrow the rows inside the join operator itself. DuckDB does it in the optimizer instead, rewriting the columns that enter the operator and leaving the operator unchanged.

Table 3.1: Surveyed techniques’ benefits and their adoption status.

3. LITERATURE STUDY

4

Small Build Side Dictionary Emission

Analytical queries often join a large table to a small one. The standard case is a star schema, where a small dimension table sits on the build side and a fact table on the probe side [36]. The join copies the dimension's columns into every fact row that matches. The dimension may hold a few hundred rows and the fact table tens of millions. The processor then spends its time copying the same dimension table values again and again.

Figure 4.1 shows one such join. A `customer` table is joined to a small `country` table to pick up each customer's country name. Ten country names sit on the build side. The probe side brings 1,500,000 customers, so that set of ten distinct values is copied about 1.5 million times.

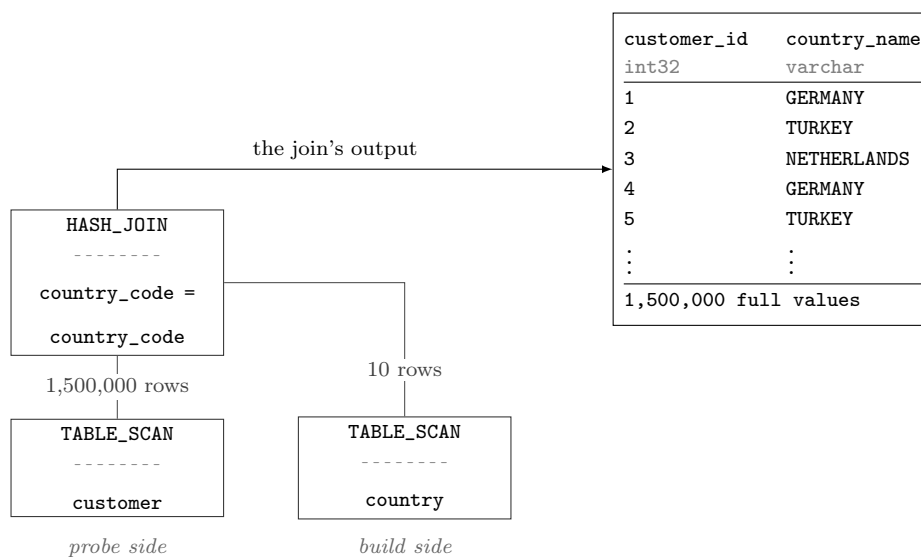


Figure 4.1: Ten country names joined to 1,500,000 customer rows. Every output row carries a copy of a name, so ten distinct values are copied about 1.5 million times.

4. SMALL BUILD SIDE DICTIONARY EMISSION

The `country_name` column holds a small set of values, repeated across many rows. This is what dictionary encoding is for [1]. This chapter makes the general hash join emit its build-side columns as dictionary vectors. The join stores each value once, and every matching output row carries a small index instead of a copy. The more rows a join emits, the more copies it saves. The wider its payload, the more it saves on each row.

The dictionaries it builds are not like the ones storage produces. The join builds them after the build side is finalized. It holds its whole input at that point, so one dictionary can cover every value the column takes. That dictionary never changes while the join runs. This is new, and we call it a *pipeline-global* dictionary. Prior work calls a dictionary that covers a whole stored column global, or column-wide [38]. This one is the same idea at runtime. An operator builds it and holds it for as long as it runs. Chapter 5 defines the guarantee and gives the join types it holds for.

The guarantee is what the next contribution needs. A dictionary that never changes can be trusted from the first chunk a consumer sees. So when this column reaches the build side of the next join, that join can store an index in every row instead of the value. The hash table is then built in compressed form, which is the main goal of this thesis and the subject of Chapter 5.

DuckDB already emits dictionary vectors from one join, the perfect hash join of Section 2.8. That comes out of the way it stores its build side, and was never an attempt to compress its output. The optimization in this chapter aims at the emission itself. It works on the general hash join, which takes keys of any type and range, duplicate keys, and the join types the perfect hash join cannot handle.

The chapter proceeds from the design to its measurement. Section 4.1 presents the approach and Section 4.2 follows one join through it, from build side to output. Section 4.3 then gives the conditions under which the optimization runs, Section 4.4 records where the change sits in the code, and Section 4.5 measures the result in isolation.

4.1 The approach

Everything up to and including `finalize` is unchanged. The join reads the build side and writes each record into the row store. In `finalize`, as Section 2.7.2 described, every record is inserted into the pointer table, and probing cannot begin until that completes. Once it does, the whole build side is known and its records sit together in the row store. At that point we add one extra pass over the records, to prepare the dictionaries.

The pass produces one array per build-side output column. The join walks the build records once, in row-store order. For each output column, it gathers that column's entry out of every record into an array, in the same order. The first record's entry lands in the first position, the second in the second, and so on. All the arrays share this order, so a given position refers to the same record in every one of them. That is what lets a single index per record stand for all of its columns. Each array is the column's dictionary. It holds one entry per record, and the output references those entries by index instead of copying a value into every row. The array has an entry for every record, not for every distinct value. A value shared by two records therefore occupies both of their entries. What an entry contains depends on the type. A fixed-width value sits in the entry directly. A string entry is a sixteen-byte header either way, in the two forms Figure 2.6 showed. A string of twelve bytes or fewer sits inside that header. The bytes of a longer one stay in the row store's heap, and the header points at them, so they are never copied. Figure 4.2 shows this pass on a small example.

The probe phase is where this pays off. Finding matches is unchanged. The engine hashes each probe key, locates the slot it belongs to, and confirms the match against the stored key as before. What changes is the step that builds the output. Instead of gathering each matched build-side column into a fresh output vector, the join emits it as a dictionary vector. The dictionary is the by-position array built after finalize; the selection vector holds one index per matched row. The output column is dictionary-encoded. Its values live once in the shared array, and the rows reference them by index rather than by copy.

Two questions follow from this design. The first is where to keep each record's index. That index is the record's position in the dictionary arrays. The probe needs it for every record it matches, so the record itself has to carry it. Recording it must not change the row store's layout, since the rest of the join depends on it. Reading it back during probing must also be fast, or the added lookups outweigh the speedup the optimization brings.

The obvious option is to add a field to each record to hold the index. That would mean altering the record layout, which is a core structure the rest of the join relies on, and enlarging every build record in the process. Changing such a central structure for one optimization is too invasive. A field that already exists would be preferable.

Such a location already exists in every record. As described in Section 2.7, each record carries a next-pointer field. During finalize it is written to link records into the chains that make up the pointer table, and during probing it is read to walk those chains when duplicate keys share a slot. Once finalize is complete, nothing writes the field again, though probing still reads it to follow the chains. When the build side has no duplicate keys, there

4. SMALL BUILD SIDE DICTIONARY EMISSION

are no chains to follow, so after finalize the field is neither written nor read again, and the index can be stored in it directly. When duplicate keys are present, the pointers are still needed, so they must be preserved before the field can be reused. The field is always present and a fixed eight bytes wide, room enough for the index. Storing the index there reuses space the record already has, without adding memory or changing the record layout.

When the build side has duplicate keys, the join saves each record's original next-pointer into an auxiliary array before overwriting it with the dictionary index, one entry per record. Chain following then takes one extra step. It reads the dictionary index from the next-pointer field, and uses that index to find the record's real chain pointer in the auxiliary array. The array is only built when duplicate chains exist. Where the build side has no chains, there are no pointers to preserve, so no array is allocated and the optimization adds no memory. Where chains do exist, the array costs eight bytes per build record. Since the optimization only runs on small build sides, this total stays modest. Figure 4.3 shows a record's layout after this step, including a duplicate chain resolved through the auxiliary array.

A chain exists only where two build records share a join key. Records whose keys merely collide under the hash are placed in a different slot rather than linked together.

The second question is when the optimization should run at all. Preparing the dictionaries has a cost, and it pays off only for certain join shapes. Section 4.3 gives the conditions under which the optimization runs.

4.2 A worked example

This section follows a single join through the whole design, from the build records to the emitted output. A store keeps a small table of country-based discounts and joins it against its large table of orders, matching each order to the discounts for its country.

```
1 SELECT o.order_id, d.label, d.code, d.percent
2 FROM orders o
3 JOIN discounts d ON o.country = d.country
```

The `discounts` table of Table 4.1 is the build side and `orders` is the probe side. A country can have more than one discount, so the join key repeats on the build side, which is what produces chains.

4.2 A worked example

position	country	label	code	percent
0	Netherlands	Seasonal	SEA10	10
1	Netherlands	Member Rate	MEMBER	5
2	Turkey	Seasonal	SEA08	8
3	Germany	Bulk Order	BULK12	12
4	Spain	Member Rate	MEMBER	15

Table 4.1: Build side: the `discounts` table. `Netherlands` appears twice, at positions 0 and 1, so it forms a chain.

order_id	country
1	Germany
2	Netherlands
3	Turkey
4	Japan
5	Spain

Table 4.2: The first five rows of the probe side, the `orders` table. The full table holds 1,000,000 orders, well above the activation threshold; these five rows are shown only to illustrate the query.

The `country` key is a string, so the perfect hash join cannot apply and the join runs on the general path. The `label`, `code`, and `percent` columns are the ones that would ordinarily be copied into the output for every matching order.

After finalize, the join gathers each of these columns into a dictionary array, one entry per discount record. The `label` array holds `Seasonal` at position zero, `Member Rate` at position one, and so on; the `code` and `percent` arrays hold their values in the same order. The three arrays share one index space, so a single position, say position two, selects `Seasonal`, `SEA08`, and `8` together. Each array has an entry per record, not per distinct value, so a value repeated across records is stored more than once. `Seasonal` occupies both position zero and position two, and the code `MEMBER` occupies positions one and four. Figure 4.2 shows this step, with the records left untouched.

Each record's position is then written into its next-pointer field, the same single index serving all three output columns. The two `Netherlands` records form a chain, so the pointer linking them is first saved into the auxiliary array. Figure 4.3 shows the record layout that results. The payload is untouched, the next-pointer field now holds the position index, and the saved chain pointer sits in the auxiliary array.

At probe time, order 1 for Germany matches the single record at position three, read

4. SMALL BUILD SIDE DICTIONARY EMISSION

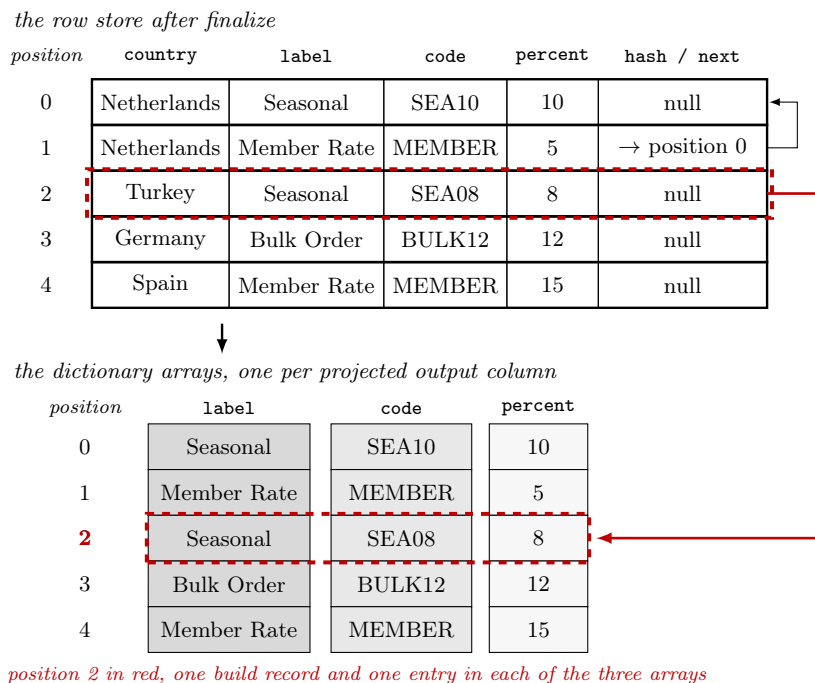


Figure 4.2: The dictionary arrays built after finalize, with the build records left unmodified. Each holds one entry per record rather than one per distinct value.

straight from that record's next-pointer field, and the output references position three. Order 2 for the Netherlands matches the chain. The head of the chain is the record at position one, since finalize adds each duplicate to the front. The join reads position one from it, then follows the saved pointer in the auxiliary array to the record at position zero and reads its index. Order 2 therefore produces two output rows, one for each Netherlands discount, referencing positions one and zero. Order 3 for Turkey matches the single record at position two. Order 4 for Japan finds no match and produces nothing.

The emitted **label**, **code**, and **percent** columns are dictionary vectors. Each has its by-position array as its dictionary, and all three share one selection vector of matched positions, one entry per output row. Figure 4.4 shows the result. A value such as **Seasonal** sits once in the **label** array and is referenced by index wherever it applies. No value is copied into the output at all.

The saving becomes clear at the scale of the full probe side. Suppose 200,000 of the 1,000,000 orders are from Turkey. Without the optimization, each of those orders copies Turkey's three build-side values into the output, which is 600,000 value copies for Turkey alone, most of them strings. With emission, Turkey's values sit once in the arrays at position two, and each of the 200,000 matching orders carries a single index that serves

the row store, with the dictionary index in the trailing field

position	country	label	code	percent	hash / next
0	Netherlands	Seasonal	SEA10	10	0
1	Netherlands	Member Rate	MEMBER	5	1
2	Turkey	Seasonal	SEA08	8	2
3	Germany	Bulk Order	BULK12	12	3
4	Spain	Member Rate	MEMBER	15	4

the dictionary arrays

position	label	code	percent
0	Seasonal	SEA10	10
1	Member Rate	MEMBER	5
2	Seasonal	SEA08	8
3	Bulk Order	BULK12	12
4	Member Rate	MEMBER	15

the auxiliary array

saved pointer
null
→ position 0
null
null
null

the index stored in row 1 selects entry 1 in the dictionary arrays and entry 1 in the auxiliary array

Figure 4.3: The same build records, with the dictionary index now in the next-pointer field and the payload untouched. The chain pointer that field held is saved first into the auxiliary array.

all three columns. The 600,000 value copies become 200,000 small index writes. The same holds for every other country, so across the whole probe side the output is built from a handful of stored values and one index per row, in place of millions of copied values.

4.3 Activation

Preparing the dictionaries is not free. The join gathers each build-side output column into a dictionary and, when duplicate chains exist, saves the original chain pointers. Those structures are allocated beside the row store rather than replacing it. Every record stays where it was, so the join carries both. This work happens once per join, after finalize, and it pays off only when the probe side is large enough that the avoided per-row copying outweighs it. When the probe side is small, or the build side is large, the preparation can cost more than it saves. The optimization therefore runs only when a set of conditions indicates that it is likely to help, and otherwise the join takes its normal gather path unchanged.

The main conditions concern the shape and size of the join. The build-side payload must

4. SMALL BUILD SIDE DICTIONARY EMISSION

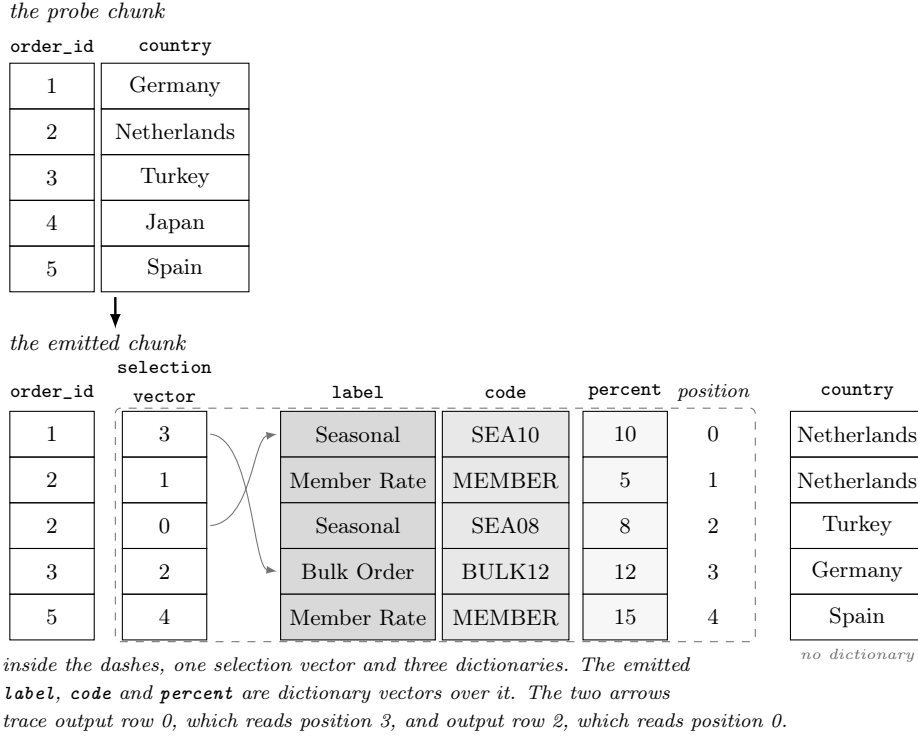


Figure 4.4: The whole join on the five orders of Table 4.2. Each matching probe row emits one output row per build record it matched, carrying a single index into the dictionaries rather than a copy of any build-side value, so **label**, **code** and **percent** share one selection vector.

be small enough that gathering it into dictionaries stays cheap and cache-resident. This is measured as a cap on total payload bytes rather than a row count, so that narrow tables may carry more rows and wide tables fewer, matching the real cost of the gather. The cap is set at 8.4 megabytes, which holds roughly a hundred thousand rows of typical dimension payloads. The probe side must also be large in absolute terms, at least 262,144 rows, since a small probe side cannot amortize the preparation. Finally, the probe side must be large relative to the build side, at least a hundred to one, since the benefit comes from each build value being reused across many probe matches.

A few further conditions rule out cases the technique cannot serve. The build side must project output columns and contain rows, or there is nothing to emit. Nested column types, such as structs, lists, and arrays, are excluded, because the dictionary representation used here does not encode them. The external, out-of-core join is excluded, since it rebuilds records from disk and invalidates the record pointers the technique relies on. Such a join has spilled, so its build side is far above the payload cap in any case. The SINGLE join is also excluded. Used for uncorrelated scalar subqueries, it emits a row for every probe

row and fills in nulls where there is no match. Its output path gathers those rows one at a time rather than emitting a dictionary vector, so the join would pay for the dictionaries without using them.

The conditions do not draw on equally reliable information. At finalize the build side has been materialized, so its row count and payload bytes are exact. The probe side has not run, so its size is the optimizer's estimate. The preparation is paid on the build side, so its cost is known exactly before the decision is taken. If the probe estimate turns out to be too high, the join has wasted that known amount and nothing more, which is what makes an estimate acceptable here.

When any condition is not met, none of this machinery runs. The join finalizes and probes exactly as described in Section 2.7, and the output is gathered row by row as before. The optimization is therefore additive. It changes the behavior of joins that qualify and leaves every other join untouched.

4.4 Implementation

The optimization lives entirely in the hash join, with the activation decision taken from the physical hash join operator.¹ The design was described above; this section records where it sits in the code and the engineering choices that keep it off the common path. The change is small. It adds 239 lines and removes 52 across three engine files, excluding tests and benchmarks.

The whole optimization enters through two places. One function runs the after-finalize preparation, building the dictionary arrays and, where duplicate chains exist, the auxiliary array. Another, at probe time, either emits dictionary vectors or falls back to the ordinary gather. Confining the change to these two entry points keeps it out of the rest of the probe code. The auxiliary array is held as an owning buffer allocated through the buffer manager, so it counts against the memory limit like any other large hash-table allocation.

The care that matters is in the chain-following accessor, since it runs in the hottest loop of the probe. It is compiled twice, once for a join that emits and once for a join that does not. The version for a join that does not emit is the original two-instruction chain walk, byte for byte what the code did before this work existed, since the unused branch is eliminated at compile time. The flag that selects between the two is read once per chunk, never per row, so the common path pays nothing at all. The join already uses this branch-elimination pattern elsewhere, for its parallel insert loop.

¹<https://github.com/duckdb/duckdb/pull/22340>

4. SMALL BUILD SIDE DICTIONARY EMISSION

The dictionaries the join emits also carry an identifier, stamped once and fixed for the life of the join. A downstream operator can therefore recognize when a chunk carries a dictionary it has already seen, instead of treating each chunk’s values as unrelated. This is what lets the emission produce pipeline-global dictionaries, the guarantee Chapter 5 defines and depends on. That chapter also gives the join types for which it holds.

4.5 Evaluation

This section measures the optimization in isolation, on micro-benchmarks built to exercise exactly the join shape it targets. These experiments establish when the optimization helps, by how much, and why, from its worst case to its best. Broader results on standard analytical workloads, and the effect of the optimization alongside the other contributions of this thesis, are reported in Chapter 7.

4.5.1 Setup and methodology

All experiments use a `VARCHAR` join key, which blocks the perfect hash join and forces the general hash-join path where the optimization lives. The payload columns hold non-inline strings, so every payload value is heap-allocated and the baseline must copy a string header into the output for every matched row. This is the case the optimization targets most directly. Build and probe sizes vary by experiment and are stated with each.

Measurements were taken on an Apple M4 Pro with 24 GB of memory. Each query runs single-threaded after two warm-up runs, under DuckDB’s profiler, with join reordering disabled so the dimension stays on the build side. Every condition is run ten times with the optimization enabled and ten disabled, toggled by a single compile-time constant. In every table below, *disabled* is the binary built with the optimization compiled out and *enabled* is the binary built with it in.

The hash join’s operator timing is where the optimization’s benefit appears. It does not include the cost. The preparation runs after the build side is finalized, between the build and probe pipelines, so the profiler attributes it to no operator. It is therefore timed in place and recorded beside the operator’s own figure. The full operator cost, used throughout, is the operator timing plus the preparation, which is the fair comparison against the disabled operator. The whole-query wall time is reported alongside it.

The single-threaded setting is a limitation. The profiler sums operator timing across threads, so under parallel execution the probe would speed up in elapsed time while the

sequential preparation would not, raising the preparation’s share and shifting the break-even points below. Single-threaded runs keep the two comparable.

4.5.2 Effect of payload width

The first experiment varies how much build-side data the join carries into its output. The build side is a 25-row dimension with 35-byte non-inline string payloads. The probe side is fixed at one million rows, all matching, and the query projects one, two, four, or eight of those payload columns. The four-column case runs as follows.

```
1 SELECT count(*), min(d.c00), max(d.c01), min(d.c02), max(d.c03)
2 FROM fact f JOIN dim d ON f.dim_key = d.dim_key
```

The aggregate is ungrouped, so it forces every projected column through the join’s output while keeping the result a single row. Having no grouping keys, it engages no dictionary optimization of its own. Widening the payload adds copy work to the baseline, since each extra column is one more string header gathered and written for every matched row. The optimization avoids this by writing one index per row that covers all of the row’s columns at once. Table 4.3 reports the full operator cost, the hash join operator timing plus the measured preparation.

payload columns	the optimization		reduction	speedup
	disabled	enabled		
1	6.275	5.792	7.7%	1.08×
2	8.454	5.773	31.7%	1.46×
4	10.427	5.857	43.8%	1.78×
8	14.767	5.986	59.5%	2.47×

Table 4.3: Effect of payload width on the full hash join operator cost (milliseconds), the operator timing plus preparation, at one million probe rows. Geometric means over ten runs.

The operator figures show the mechanism at work. Without the optimization, the hash join grows with the number of payload columns, from 6.3 to 14.8 milliseconds as the projection widens from one to eight columns, since each added column is one more header copied per matched row. With the optimization the operator cost is almost flat, between 5.8 and 6.0 milliseconds at every width. The per-row cost has become a single index read from the next-pointer field, and that one index serves every output column, so adding a column adds only a zero-copy dictionary vector per chunk rather than a copy per row. The

4. SMALL BUILD SIDE DICTIONARY EMISSION

saving therefore widens as the payload widens. The operator speedup climbs from 1.08 times at one column to 2.47 times at eight, driven entirely by the growth of the baseline while the optimized cost holds constant.

The full operator cost includes the preparation, and here it is negligible. Over a 25-row build side the preparation takes a few microseconds, growing from about 2.6 to 5.7 microseconds across the four widths. It is two to three orders of magnitude smaller than the operator saving it buys, so the full operator figures are indistinguishable from the timing alone.

The whole-query figures of Table 4.4 are smaller, for two reasons. First, the join is only one part of the query. It runs alongside the probe-side scan and the operators that consume its output, none of which the optimization changes, so its saving is a small part of the total wall time. Second, the join emits its build-side columns as dictionary vectors, but the `UNGROUPED_AGGREGATE` that consumes them is not dictionary-aware. It works once per row rather than once per distinct value, so the join’s compression saves it nothing. The full end-to-end gain arrives only when downstream operators consume the dictionary vectors directly, without flattening them, which is the subject of later chapters. This effect recurs in every experiment below, and later sections refer back to it rather than restating it.

payload columns	the optimization		reduction speedup	
	disabled	enabled		
1	17.259	16.746	3.0%	1.03×
2	24.730	21.884	11.5%	1.13×
4	34.740	30.192	13.1%	1.15×
8	55.009	46.948	14.7%	1.17×

Table 4.4: Effect of payload width on the whole-query wall time (milliseconds) at one million probe rows. Geometric means over ten runs.

The whole-query speedup still grows with payload width, from 1.03 times at one column to 1.17 times at eight. At one column the gain is slight but real, the enabled runs faster at every width. From two columns on it is clear and consistent. The operator, by contrast, is already faster at one column.

4.5.3 Effect of total rows emitted

The previous experiment fixed the probe side and varied the payload. This one fixes the payload and asks what property of the probe side drives the saving. The answer is the number of rows the join emits. Each emitted row replaces one string-header copy per

4.5 Evaluation

column with a single index read, while a probe row that finds no match emits nothing and costs the optimization nothing. The probe side sets a ceiling on how many rows can emit without setting the saving itself.

Both sweeps use the same 25-row build side, fix the payload at four non-inline string columns, and report the full operator cost. The build keys are unique, so each probe row emits at most one output row. A build side with duplicate keys would push the emitted count above the probe count, since a probe row then matches every record in its chain. Emitted volume was varied in two ways, by changing how many probe rows find a match over a fixed ten-million-row probe, and by changing the probe size with every row matching.

Table 4.5 reports the first sweep. The absolute saving tracks emitted volume closely. Five million emitted rows save 23.59 milliseconds, seven and a half million save 31.97, and ten million save 45.72, which is between four and five milliseconds for every million rows emitted.

probe rows	emitted rows	per chunk	the optimization		reduction
			disabled	enabled	
10,000,000	5,000,000	1,024	65.32	41.73	36.1%
10,000,000	7,500,000	1,536	83.34	51.37	38.4%
10,000,000	10,000,000	2,048	106.42	60.70	43.0%

Table 4.5: Effect of total rows emitted on the full hash join operator cost (milliseconds) over a fixed ten-million-row probe with four payload columns. The per-chunk column gives the rows emitted from each 2,048-row probe chunk. Geometric means over ten runs.

The reduction as a percentage moves as well, from 36.1 to 43.0 percent across the same three volumes. The saving per emitted row is close to constant, so that movement comes from elsewhere.

DuckDB probes in chunks of up to 2,048 rows. Some of the work of walking a chunk is spent whether a given row emits or not, so a chunk in which few rows emit still pays it. The emission removes work only from the rows that emit. Its share of the chunk’s total work should therefore grow as more of the chunk emits. That is what these three conditions show. At five million total emitted rows each chunk emits 1,024 of its 2,048 rows, at seven and a half million it emits 1,536, and at ten million every row of the chunk emits.

Table 4.6 separates the two quantities. Every condition in it emits a full chunk of 2,048 rows, so the only thing that changes is the total. That total spans a hundredfold range,

4. SMALL BUILD SIDE DICTIONARY EMISSION

from five hundred thousand rows to fifty million, and the absolute saving spans the same range. The reduction holds between 42.3 and 43.0 percent throughout.

probe rows	emitted rows	per chunk	the optimization		reduction
			disabled	enabled	
500,000	500,000	2,048	5.20	3.00	42.4%
5,000,000	5,000,000	2,048	53.23	30.72	42.3%
10,000,000	10,000,000	2,048	106.42	60.70	43.0%
50,000,000	50,000,000	2,048	525.03	302.30	42.4%

Table 4.6: Full hash join operator cost (milliseconds) with every probe row emitting, so each chunk emits its full 2,048 rows, across a hundredfold range of total emitted rows. Four payload columns. Geometric means over ten runs.

Probe size multiplies the magnitude of the saving without moving its rate. A join with a large probe and a modest emitting fraction can still emit enough to pay. A probe too small to emit much is excluded by the activation conditions of Section 4.3, which require at least about a quarter of a million probe rows.

The same measurements settle the cost side. The dictionary build is paid once per join, and it measured between four and five microseconds in every condition, whether the probe held five hundred thousand rows or fifty million. It depends on the build side alone, which is 25 rows throughout. Set against savings measured in milliseconds, any join emitting at these volumes repays it many times over.

4.5.4 Worst case

Section 4.5.3 established that the saving is proportional to the rows a join emits. This subsection asks what happens when a join qualifies for the optimization and then emits almost none. The optimization pays a fixed preparation cost after the build side is finalized, building a dictionary array for every projected column and, when duplicate keys form chains, allocating and filling the auxiliary array. That cost is paid before a single probe row is seen. A join that emits few rows has almost nothing to repay it with. The activation conditions cannot prevent this, since they are applied before the probe runs and cannot know how many rows the join will go on to emit.

To find the bound, the setup is made as hostile as the activation conditions allow. The build side has 26,000 rows and projects twenty non-inline string columns, which brings the counted payload to just under the 8.4-megabyte cap, the largest preparation the conditions permit. Its keys are duplicated so that chains form and the auxiliary array is allocated,

4.5 Evaluation

three build rows to a key, so every probe row that matches emits three output rows. The probe side is 2.7 million rows, the smallest that satisfies the hundred-to-one ratio at this build size. Holding all of that fixed, emitted volume alone is varied, starting from zero, where the preparation is wasted in full. Matching and non-matching probe rows are interleaved uniformly, so every execution chunk emits the same number of rows.

The three-number measurement of Section 4.5.1 is what makes this case clear. The preparation cost is measured directly, so the operator can be charged its full cost, the operator timing plus the preparation, rather than only its probe-side benefit. Across the sweep the preparation is a near-constant one millisecond, since it depends on the build side, which does not change. Table 4.7 reports the full operator cost and Table 4.8 the whole-query wall time.

probe rows	match rate	matched per chunk	emitted rows	the optimization		difference
				disabled	enabled	
2,700,000	0%	0	0	2.76	3.78	+37.2%
2,700,000	5%	102	404,979	12.02	8.30	−31.0%
2,700,000	10%	205	809,964	19.43	10.64	−45.2%

Table 4.7: Worst-case setup: full hash join operator cost (milliseconds), the operator timing plus the measured preparation. Each key holds a chain of three build rows, so every probe row that matches emits three output rows, which is why the emitted total runs to three times the matched rows. Geometric means over ten runs; a positive difference is a regression.

probe rows	match rate	matched per chunk	emitted rows	the optimization		difference
				disabled	enabled	
2,700,000	0%	0	0	4.80	5.89	+22.6%
2,700,000	5%	102	404,979	88.43	90.00	+1.8%
2,700,000	10%	205	809,964	167.73	144.10	−14.1%

Table 4.8: Worst-case setup: whole-query wall time (milliseconds) over the same conditions. The three-row chains again put the emitted total at three times the matched rows. Geometric means over ten runs; a positive difference is a regression.

With nothing emitted, the preparation is pure waste. It is a single fixed cost of about one millisecond, spent building twenty dictionary arrays and the auxiliary array and then thrown away in full. The hash join operator’s own timing moves by a fraction of a percent, because the probe does the same work whether the optimization is present or not when no row matches. The full operator cost regresses by 37.2 percent purely because that one

4. SMALL BUILD SIDE DICTIONARY EMISSION

millisecond of preparation is added on top, and the whole query, only a few milliseconds long at zero output, regresses by 22.6 percent. The regression at the worst case is exactly the preparation, and nothing else.

The preparation is fixed and every emitted row is a saving, so the regression is repaid in proportion to emitted volume. Each emitted row avoids twenty string-header gathers and pays one index read instead, a saving that measures at a near-constant twelve nanoseconds per emitted row across the sweep. The operator saving therefore scales linearly with emitted rows while the preparation stays near one millisecond, so the preparation is recovered after roughly eighty thousand emitted rows. The full operator cost reflects this. From a 37.2 percent regression at nothing emitted it turns to a 31.0 percent saving at 404,979 rows and a 45.2 percent saving at 809,964. Eighty thousand rows is well below the lowest emitting condition measured, which is already 31.0 percent ahead, so the operator carrying the optimization's full cost recovers its preparation long before the sweep's first step.

The whole query crosses over later. The wide twenty-column output is materialized downstream whether or not the optimization is present, and that shared cost grows with emitted volume, so it masks the join's saving in the wall time. The whole-query difference moves from plus 22.6 percent with nothing emitted, to plus 1.8 percent at 404,979 rows, to minus 14.1 percent at 809,964, so the crossover falls between those two conditions. The operator is already winning well before the query as a whole is.

This is a constructed worst case, not a workload anyone runs. It projects twenty columns from a dimension join, forces the keys into chains, pushes the payload to the activation cap, and produces an output wide enough to be memory-bound. A join that matches most of a large probe emits far more than this setup needs to turn positive. The regression is therefore confined to joins that emit few rows, and bounded within them. At its worst, with nothing emitted at all, it costs one millisecond of wasted finalize work.

4.5.5 Best case

This final experiment measures the optimization on the shape it is built for, a small build side, a wide non-inline payload, and a large fully matching probe. The build side is ten rows with unique keys, so no chains form, the auxiliary array is never allocated, and the preparation reduces to gathering twenty ten-entry arrays and writing one index per row. The probe side is ten million rows at one hundred percent match, so every probe row emits and receives the full per-row saving across all twenty projected columns. The projection is twenty columns wide, as in the worst case, which makes the per-row saving as large as it can be.

Table 4.9 reports the whole-query wall time and the full operator cost.

metric (ms)	the optimization		reduction	speedup
	disabled	enabled		
whole-query wall	970.70	818.85	15.6%	1.19×
full operator	216.97	61.12	71.8%	3.55×

Table 4.9: Best-case setup: whole-query wall time and full hash join operator cost (operator timing plus preparation) at ten million probe rows, one hundred percent match, twenty non-inline payload columns over a ten-row build side. Geometric means over ten runs.

The hash join operator drops from 217 to 61 milliseconds, a 71.8 percent reduction and a 3.55 times speedup, the largest operator win in the evaluation. The arithmetic follows the mechanism directly. Ten million emitted rows across twenty columns are two hundred million string-header gathers on the baseline; the optimization replaces them with ten million index reads, one per row, shared by all twenty columns. The 156 milliseconds saved work out to about sixteen nanoseconds per emitted row, or under one nanosecond per avoided header copy. This is close to the largest per-emitted-row saving the optimization can show, a wide projection at full emission over a large probe.

The preparation, measured directly, is about ten microseconds. With unique keys there is no auxiliary array and no chain-pointer snapshot, and the build side is only ten rows, so the preparation reduces to gathering twenty arrays and writing one index per row. It is some fifteen thousand times smaller than the saving it unlocks, so the full operator cost is indistinguishable from the operator timing. A negligible microsecond-scale setup buys a 3.55 times operator speedup.

The whole-query win is smaller than the operator win, at 1.19 times, for the two reasons given earlier. The aggregate that consumes all ten million rows across twenty columns is the query’s dominant cost, and emission does not accelerate it. It also receives the columns as dictionary vectors and reads them back through the index rather than using that form, so the compression is undone as soon as the join hands its output on. The 3.55 times speedup is real and lives in the hash join. The operator result is the one that isolates what the optimization does, and there it delivers its largest margin.

4. SMALL BUILD SIDE DICTIONARY EMISSION

5

The Dictionary-Surviving Hash Join

Memory is expensive, and getting more expensive. Conventional DRAM contract prices rose by about 93 to 98 percent in a single quarter at the start of 2026 [52], and memory is expected to take a growing share of what a server costs [57]. One of the largest structures an analytical query builds is a join’s hash table. It materializes every build row at once, so it often sets the query’s peak memory [31]. Building that table in compressed form is the main goal of this thesis, since a smaller hash table can lower a query’s peak memory.

Figure 5.1 shows a query that totals revenue by country. It reads a ten-row `country` table, 1,500,000 customers, and 15,000,000 orders, and groups on the country name. DuckDB plans it as two joins. The first attaches each customer’s country name. The second attaches those customers to their orders.

The previous chapter’s emission makes the first join emit `country_name` as a dictionary vector rather than copying a name into every output row. That column then flows straight into the build side of the second join, which is the case this chapter targets. The second join writes 1,500,000 rows into its hash table, materializing every row of the column on the way in. This chapter presents the dictionary-surviving join, which builds that hash table in compressed form instead. Not every dictionary is suitable for the job.

The chapter proceeds from the design to its measurement. Section 5.1 defines the dictionary guarantee and Section 5.2 the layout decision the join defers. Section 5.3 presents the mechanism and Section 5.4 follows one column through it. The rest covers the operator in between that used to destroy the encoding, the conditions under which the join activates, where the change sits in the code, and a measurement in isolation.

5. THE DICTIONARY-SURVIVING HASH JOIN

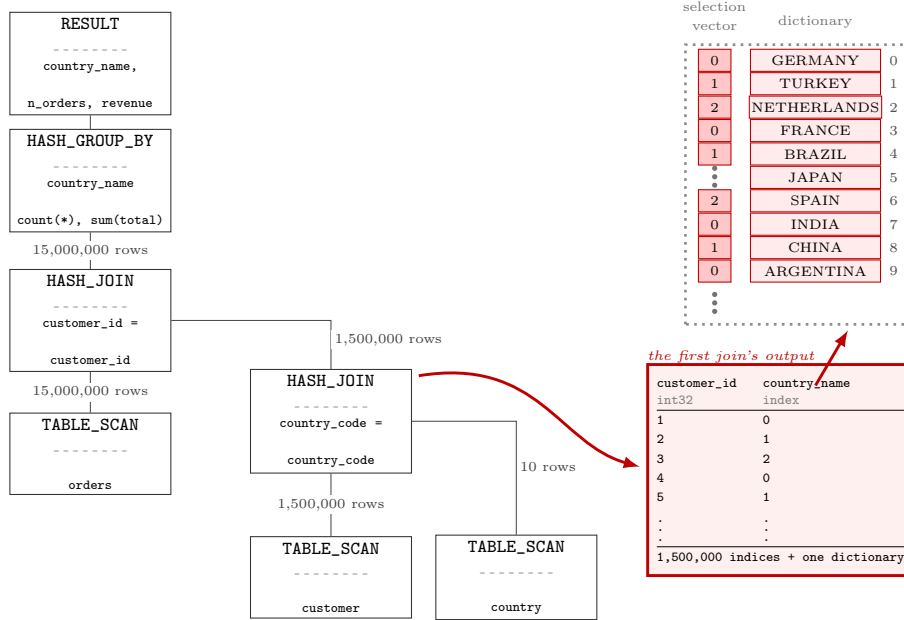


Figure 5.1: The example query. The first join emits `country_name` as a dictionary vector, so the chunks arriving at the second join’s build side carry one index per row over a single dictionary.

5.1 Pipeline-global dictionaries

We introduce a new concept, the *pipeline-global dictionary*. It is the main conceptual contribution of this thesis.

Prior work classifies a dictionary by its scope, meaning how much data one dictionary describes. Łuszczak names three, partition-wide, column-wide and domain-wide [38]. A column-wide dictionary covers a whole column, and most of the literature calls such a dictionary global. Column-wide scope is enough for the join to keep a payload column compressed in its hash table. However, a join key has to be compared against the probe side, so both sides must agree on one encoding, which requires domain-wide scope [38]. A payload column is never compared. That means a column-wide dictionary is safe for storing a payload column in the hash table in compressed form.

A global dictionary is normally a storage-side structure, built over a column’s stored values rather than during query execution [1, 31]. Such dictionaries have seen limited adoption, since they are expensive to maintain while the column underneath keeps changing. Every insert, update and delete has to be reconciled with it, which can mean re-coding rows or collecting garbage. The dictionary must also stay in memory in full, and amounts to a mandatory secondary index on every string column [31]. Łuszczak reports the same

5.1 Pipeline-global dictionaries

difficulties from concurrent transactions, and adds that one dictionary over a whole column compresses worse than several smaller ones when a column’s values are clustered [38]. Adding a value the dictionary has not seen can be the most expensive operation of all, because once the codes need one more bit, every value already stored has to be compressed again [61].

A pipeline-global dictionary is global in the same sense, since it covers every row of the column, but it differs from those above in where it comes from. A hash join builds it during execution, over rows it has already materialized, so no insert or update can reach it and it never has to grow. Much of the overhead above follows from a changing column, so it does not arise for a dictionary built this way. The producer holds it for its whole life and wraps the same dictionary on every chunk. A consumer can therefore commit to it on the first chunk it sees.

Two structures in the literature are also built during execution rather than over stored data. Gubner et al.’s Unique Strings Self-aligned Region is created anew for each query by inserting frequently occurring strings as the query runs [31]. Afroozeh ported it into DuckDB as the Unified String Dictionary, where an optimizer rule places an insertion operator below the nearest materializing operator that handles strings [4]. Neither can be committed to in advance, because the region is fixed in size, so insertion fails once it is full and no operator may assume that a given string is in it [31, 4]. That makes a hash or a comparison cheaper without letting an operator fix a row layout before the data arrives. The DuckDB port also takes only strings longer than twelve characters, since a shorter one already sits inside the row that holds it [4].

The pipeline-global dictionary is therefore a scope the taxonomy above does not name. It belongs to one materializing operator rather than to any unit of stored data.

Storage gives no such guarantee. It builds a dictionary per row group, which changes as the scan crosses from one row group to the next. Not every row group is dictionary-encoded either. Dictionary encoding is one of several schemes DuckDB can choose for a row group [16]. A column may therefore arrive as a dictionary in one row group and in another form in the next. Figure 5.2 puts the two kinds side by side.

5. THE DICTIONARY-SURVIVING HASH JOIN

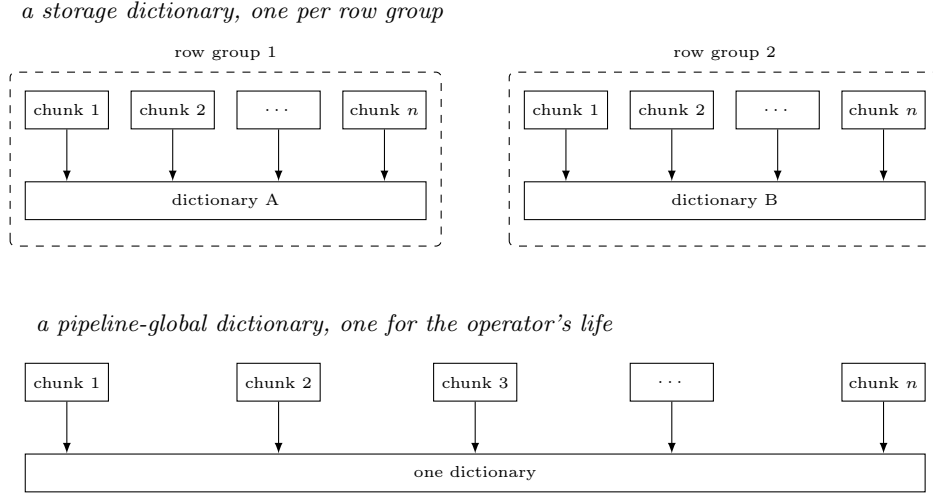


Figure 5.2: A storage dictionary against a pipeline-global one. Storage builds a separate dictionary per row group, so it changes at every boundary. A row group holds roughly sixty chunks, which the figure compresses. A pipeline-global producer wraps one dictionary on every chunk for as long as it lives.

A hash join’s build side fixes its row-store slot widths on the first chunk, before it writes a single row. If it decides on a narrow index there, that decision holds for the whole join, and what it writes into each row is an index into one particular dictionary. A single chunk is no evidence about later chunks. If the sink guesses wrong, it writes indices for one dictionary and reads them back against another, so the join emits values that were never in its input. The pipeline-global guarantee is what rules that out.

The guarantee is a single condition. Every chunk the producer emits for that column is a dictionary vector over one and the same dictionary, under one non-empty identifier, for as long as the operator lives. It admits no exception, since a producer that sends a flat vector or a constant vector on even one chunk leaves the sink a chunk it cannot read as indices.

The identifier is what lets a consumer recognize a dictionary it has already seen, so a cache keyed on it stays valid.

Two operators in DuckDB satisfy it. The perfect hash join allocates one dictionary per output column when it builds its table, then wraps that dictionary on every probe chunk, which is its only path. The small build-side dictionary emission of Chapter 4 materializes its arrays once at finalize, then wraps them for every matched chunk. It holds the guarantee for every join type whose output rows all come from matches. A `LEFT` or `OUTER` join also emits unmatched probe rows, whose build-side columns it fills with a constant null vector,

so its dictionary is not pipeline-global.

A storage dictionary and a pipeline-global one are indistinguishable on arrival, since both are dictionary vectors and both carry a non-empty identifier. What a consumer cannot see from one chunk is that a storage identifier will not last. Only the producer knows which kind it built, so the producer declares it. A single boolean on the dictionary entry records that the guarantee holds, spelled `global_dictionary` in the code. It defaults to false, and only the two qualifying producers set it, so a storage dictionary can never be admitted by accident.

5.2 Deferring the layout decision

Every slot in a build row takes its width from a `TupleDataLayout`, and the hash table built that layout in its constructor, from the columns' logical types alone. A `VARCHAR` payload, for example, reserves sixteen bytes there, and nothing later can change it.

The information needed to narrow the slot does not exist when the constructor runs. Whether a column arrives as a pipeline-global dictionary, and how many entries it has, depend on the data. The build pipeline creates its sink state, and with it the hash table, before it has produced a single chunk. Narrowing the slot requires knowing the dictionary, knowing the dictionary requires a chunk, and the chunk arrives well after the layout is fixed.

The solution is to move the decision to the first chunk. We split the hash table's initialization in two phases. The constructor keeps only the work that does not depend on the layout, and everything that does is deferred to a second phase, which runs once a build chunk has arrived and the dictionary sizes are known. Until then the hash table has no row store at all.

As Section 2.7.2 described, several threads consume the build side at once. Each has a hash table of its own and scatters its rows into its own row store, and those row stores are merged when the build ends. Whichever thread sees a chunk first therefore decides the layout, and it publishes that layout for the others to use rather than letting each build its own. Every thread then writes its rows at the same byte offsets, so the merge splices their row stores together without rewriting a row.

The slot width follows from the dictionary's size. A stored index is a value in $[0, D - 1]$, so a dictionary of at most 256 values fits in one byte, at most 65,536 in two, and anything larger in four. A column is narrowed only when its index is strictly narrower than its native slot. This is the never-regress rule, and it means the encoding can only shrink a

5. THE DICTIONARY-SURVIVING HASH JOIN

row. Strings are not the only type that benefits. A **BIGINT** column drawn from a dictionary of at most 256 values narrows from eight bytes to one. Even a four-byte **INTEGER** shrinks the same way, and a **SMALLINT** narrows from two bytes to one.

5.3 The approach

With the layout decision moved to the first chunk, the join publishes a narrowed layout on that chunk, scatters indices as it writes its rows, and rebuilds the dictionary vector at the probe. None of these three steps requires a change to the row store itself.

On the first build chunk, the publisher examines each payload column. A column that arrives as a pipeline-global dictionary, and whose index would be narrower than its native slot, is admitted, and the published layout gives it an index slot of one, two, or four bytes. Every other column keeps its native width. At the same time the join pins the dictionary of each narrowed column and holds it for the whole join, storing the stable identifier with it. The pin is a deep copy the join owns, not a reference to the producer's dictionary, because the producer's heap-allocated non-inline strings are recycled before this join probes and a reference would dangle. A stored index in the row layout means nothing on its own. It only resolves against this copy. Indices are read back during the probe, so the copy has to outlive the build and stay alive to the last probe chunk.

Every build chunk then scatters indices rather than values. A dictionary vector already stores, for each row, the index of that row's value in the dictionary, and those indices are what the join writes into the narrowed slots. The row store never knows that a dictionary was involved. It sees a column of plain integers and copies them like any other. Only the index is written; the value itself is never copied.

At probe time the join rebuilds the dictionary vector. For each matched row it reads the narrow index back out of the row store. The indices of a chunk's matched rows form a selection vector. A dictionary vector is a selection vector paired with a dictionary, and the join now has both, so it assembles one directly. The join emits it with the same stable id it received, so the dictionary survives the join under its own identifier. No value is copied.

This survival is what gives the dictionary-surviving join its name. The column keeps the same stable identifier on the way out, so dictionary-aware operators downstream can recognize it and work on the compressed column, rather than a materialized one.

5.4 A worked example

This section follows the `country_name` column through the second join of the example query. Its query plan is the one Figure 5.1 draws, and Figure 5.3 gives the three tables it reads.

```

1 SELECT n.country_name, COUNT(*), SUM(o.total)
2 FROM country n
3 JOIN customer c ON c.country_code = n.country_code
4 JOIN orders o ON o.customer_id = c.customer_id
5 GROUP BY n.country_name;

```

country		customer	
country_code	country_name	customer_id	country_code
varchar	varchar	int32	varchar
DE	GERMANY	0	DE
TR	TURKEY	1	TR
NL	NETHERLANDS	2	NL
⋮	⋮	⋮	⋮
10 rows		1,500,000 rows	

orders		
order_id	customer_id	total
int64	int32	decimal
0	0	0.00
1	1000003	0.01
2	500006	0.02
⋮	⋮	⋮
15,000,000 rows		

Figure 5.3: The three tables the example query reads, with the first rows of each and the row counts the rest of this section uses.

The first join matches on a country code rather than on a small dense integer, which blocks the perfect hash join and leaves the general hash-join path. A ten-row build side under a probe of 1,500,000 customers activates the small build-side dictionary emission of Chapter 4, so that join emits `country_name` as a pipeline-global dictionary of ten entries. That column then becomes a build-side payload of the second join, which is the one traced here. It does not arrive from storage. It arrives already encoded, each customer row carrying an index into the ten country names, as Figure 5.4 shows.

The first chunk to reach the sink decides the layout. Its `country_name` column is a dictionary vector whose dictionary is flagged pipeline-global and holds ten entries, so a

5. THE DICTIONARY-SURVIVING HASH JOIN

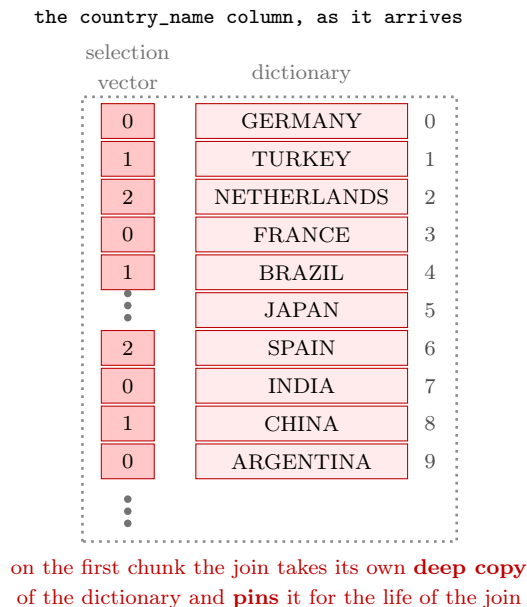


Figure 5.4: The `country_name` column as it reaches the second join’s build side, one index per row over a ten-entry dictionary. The join takes a deep copy of that dictionary rather than a reference, since the producer’s strings are recycled before this join probes.

one-byte index suffices. One byte is narrower than the sixteen a `VARCHAR` slot takes, so the column is admitted. The publisher builds a layout giving `country_name` a one-byte slot and shares it with the other build threads. The join copies the ten country names into memory of its own and records the dictionary’s identifier.

Every build chunk is then scattered under that layout. The row for customer 3 stores the byte 2 where it would have stored a sixteen-byte string slot holding `NETHERLANDS`. No row store in this join ever holds a country name. Figure 5.5 sets the two forms of the table side by side.

The probe side is unchanged. Order rows are hashed and matched on `customer_id` exactly as before, since the optimization touches only payload columns. When an order matches customer 3, the join reads that row’s payload and finds the byte 2 rather than a string. The indices of all matched rows in a chunk form a selection vector, which the join pairs with its ten country names to build a dictionary vector. Figure 5.6 shows what leaves the join.

Nothing is materialized on the way out. What the join emits is the selection vector of matched indices paired with the dictionary it pinned on the first build chunk, under the identifier that dictionary arrived with. The column therefore leaves the join in the form

5.4 A worked example

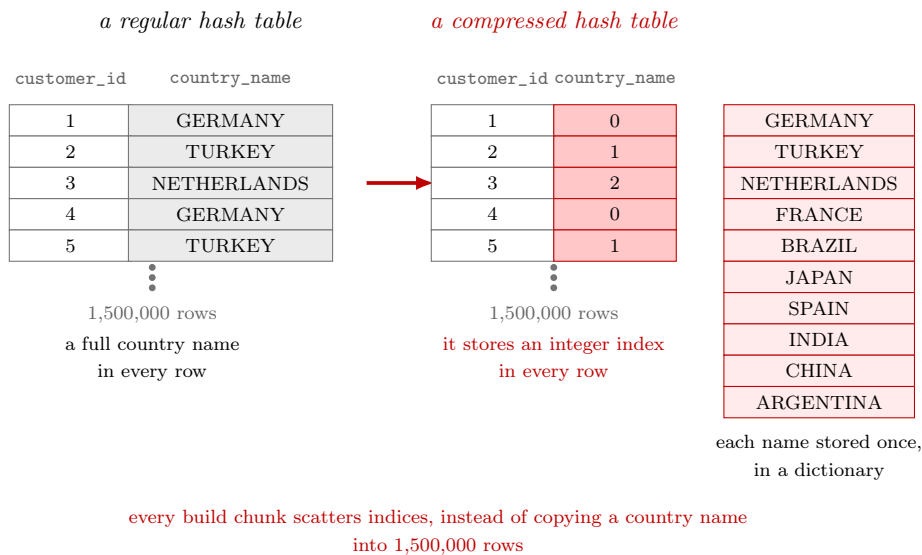


Figure 5.5: The same build side, stored both ways. On the left every row holds a full country name. On the right it holds a one-byte index, and the ten names sit once in the dictionary the join pinned.

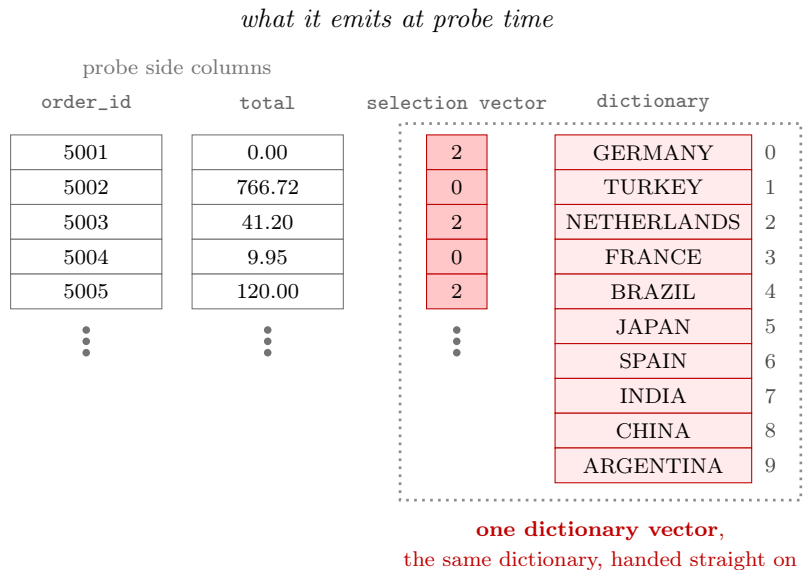


Figure 5.6: What the join emits. The probe row's own columns pass through, and country_name leaves as a selection vector over the dictionary the join pinned, under the identifier it arrived with.

it entered. The dictionary survives the join under the same identifier, which is what gives the dictionary-surviving join its name.

Across the whole build side the saving is the narrowed slot multiplied by the rows. The

5. THE DICTIONARY-SURVIVING HASH JOIN

`country_name` slot falls from sixteen bytes to one over 1,500,000 rows, so that column's share of the row store drops from 24.0 megabytes to 1.5. The ten names are stored once rather than 1,500,000 times.

The same saving repeats where the join emits, since it produces 15,000,000 output rows and on the ordinary path each of them copies a country name. Emitting a dictionary vector writes an index instead, so 240 megabytes of string headers become 60 megabytes of indices and 15,000,000 copies are never made.

The next dictionary-aware operator can therefore work on the compressed column rather than a materialized one. Here that operator is the hash group-by, which is dictionary-aware in DuckDB already, resolving each dictionary entry once and reusing the answer for the rows sharing it [22]. It is the same operator as before, now running on compressed data.

5.5 Preserving dictionaries through the caching operator

Other operators can sit between the producer of a pipeline-global dictionary and the consumer that relies on it. Every hash join's probe output passes first through a caching operator. DuckDB moves data between operators in chunks of up to 2,048 rows, and the cost of handling a chunk is largely fixed, a virtual call and a loop preamble paid once however many rows the chunk carries. A full chunk spreads that cost over all its rows. A selective join emits many small chunks instead, and each one pays nearly that same fixed cost for a fraction of the work. The operator accumulates them until it can pass on one full chunk. It promises its consumer the same rows in the same order, regrouped into fuller chunks. It promises nothing about how a column is physically represented.

The caching operator was type-blind. It allocated its accumulator from the chunk's logical types, which carry no information about a column's physical encoding, so it built a flat accumulator, and appending flattened every value. On flush it emitted a flat column. It does not take every chunk, though. A chunk larger than a size threshold passes straight through untouched, since coalescing it would buy nothing. A selective join's output therefore reached the sink as a mixture, large chunks still carrying their dictionary and coalesced ones arriving flat.

That mixture is what broke the guarantee. The producer wrapped one dictionary on every chunk, and the guarantee held when the chunk left it. An operator in between then changed the encoding on some chunks and not others, so the sink was handed a stream that violated it after the producer had done everything right. The sink had no reason to trust any chunk, and no way to see that an operator in between was the cause.

We therefore changed the caching operator to preserve what it is given. On the first append into an empty accumulator it inspects each column. When a column arrives as a dictionary under a non-empty identifier, the operator pins its dictionary entry and concatenates each later chunk's indices into one accumulated selection vector. A dictionary with no identifier is refused, since a producer that mints a fresh entry for each chunk leaves nothing to pin. On flush it re-wraps the column over that entry, so the column leaves under the identifier it arrived with and the guarantee travels with it. Columns arriving flat are appended flat, as before, and every column's rows stay aligned.

On each later chunk the operator checks whether a column's dictionary is still the one it started with. If a dictionary that made no such promise has changed, the operator gives up on that column, flattens it, and carries on. If a pipeline-global one has changed, something is wrong, since its producer promised it would not. The operator raises an error rather than pass a corrupted row store downstream.

With that change the caching operator preserves any dictionary that carries an identifier. The guarantee survives the operator in between, which is what lets a sink commit its layout on the first chunk.

5.6 Activation

The optimization is decided on the first build chunk. The join as a whole must be eligible, and then each payload column is considered on its own.

A join is eligible when its build side comes from a single source. A `UNION ALL` or a recursive CTE feeds one sink from two producers, each guaranteeing only its own chunks. The sink could then scatter indices into a column with no dictionary behind them. A single walk of the build subtree settles this when the sink state is built. It reads the physical plan the optimizer already produced, so its answer cannot change while the query runs.

Three join types are excluded as consumers, for two reasons. `LEFT` and `SINGLE` have a gather path that cannot decode an index. A `LEFT` join with unique build keys, and a `SINGLE` join, gather each payload column straight into a flat output vector. That path never reaches the branch that decodes, so it would read a narrowed slot at the column's native type. `OUTER` has an output stream that is not uniform. Its matched rows pass through the dictionary-aware gather, while its unmatched probe rows arrive as a separate chunk whose build-side columns are a constant null vector. That stream carries a dictionary on some chunks and a constant vector on others, which breaks the guarantee for whatever

5. THE DICTIONARY-SURVIVING HASH JOIN

consumes it. The join must also project at least one build-side column, or there is nothing to narrow.

The **OUTER** exclusion follows from how unmatched rows are filled today rather than from the design. A dictionary vector takes its nulls from the entry each index selects, so reserving one null entry and pointing every unmatched row at it would keep the stream uniform. A **LEFT** join reaches the same fill when its build keys are not unique, so the sentinel would cover that half of it. The exclusion is decided per join type and the sink cannot tell the two halves apart, so both causes have to be fixed before **LEFT** can be admitted.

The perfect hash join reads payload columns at their native width, so a narrowed index would be misread. Whether a join uses it is decided at finalize, after the layout was fixed. The sink therefore relies on the perfect-hash-join check made when the sink state is constructed, and a join that might use it does not use this optimization. That check can be wrong in the permissive direction. A build key carrying no statistics fails it, so this optimization publishes a narrowed layout, and the minimum and maximum computed at finalize can then bring the perfect hash join back into play. Finalize therefore looks again and abandons it when a narrowed layout is already published. The two paths never share a hash table, so whichever commits first runs. A join that takes the perfect hash join still emits its payload columns as pipeline-global dictionaries, so what it gives up is the narrowed row store rather than the dictionary.

The small build-side dictionary emission of Chapter 4 works on the same columns, so it cannot run on a join that uses this one. The two commit at different times, this optimization on the first build chunk and the emission not until finalize. A column narrowed here cannot be emitted.

Four conditions apply per column. The column must arrive as a dictionary vector flagged pipeline-global. Its type must not be nested, since a struct, list, or array cannot be wrapped as a dictionary in this form. It must not be read by a residual predicate, the non-equality remainder of a join condition. Such a predicate gathers the build column straight from the row store, with no opportunity to decode. Its index must be strictly narrower than its native width, which is what keeps the optimization from enlarging a row.

Unlike the emission, an external join is not excluded. The join's copy of the dictionary lives in memory and is never spilled. A narrow index is an ordinary row-store column, so a join that spills and repartitions moves whole rows under the same narrowed layout, and both the rows and the dictionary survive.

When any condition fails, none of this machinery runs. The layout is published with native widths, the scatter and gather take their unmodified paths, and the join behaves as it did before.

5.7 Implementation

The optimization spans the hash join and two structures around it. The dictionary vector carries the guarantee, and the caching operator must preserve it.¹ The design was described above; this section records where it sits in the code and the engineering choices behind it. The dictionary-surviving join adds 693 lines and removes 53 in ten engine files. The caching operator’s generalization adds a further 49 and removes 29 in two of them. Both figures exclude tests and benchmarks. Together they are the largest footprint of the three contributions.

Publishing the layout has to be guarded, since several threads can reach the sink at the same time. Exactly one thread builds and publishes it while the others wait, and every thread after that reads the published layout without synchronizing at all. A join whose build side is empty never publishes, so finalize publishes a native-width layout under the same guard before anything reads the row store. Figure 5.7 traces one build chunk through the gate and gives the reason for each check.

The scatter and gather changes are pre-passes rather than rewrites. Before a chunk is written, a narrowed column’s dictionary vector is replaced by a flat vector of its indices, which the existing serialization then writes as it would any integer column. The row store’s scatter code needs no change at all. At the probe a matching pre-pass reads the indices back out of the matched rows and re-emits the dictionary vector over the held copy. The matching path is untouched throughout, since only payload columns are affected.

The join’s copy of the dictionary is a deep copy. Both pipeline-global producers fill their dictionary with a zero-copy gather, so a string too long to store inline keeps a bare pointer into the *producer’s* row-store heap. That pipeline finishes and recycles its heap long before this join probes. A pin holding only a reference would leave every long string dangling. The join therefore allocates its own storage and copies the entries into it, keeping the producer’s identifier so that downstream caches still key correctly. The copy costs a few kilobytes for a dimension table’s worth of strings.

Two conditions cannot be answered from a chunk, whether the build side has multiple sources and whether the join might use the perfect hash join. Both are computed once

¹<https://github.com/duckdb/duckdb/pull/23360>

5. THE DICTIONARY-SURVIVING HASH JOIN

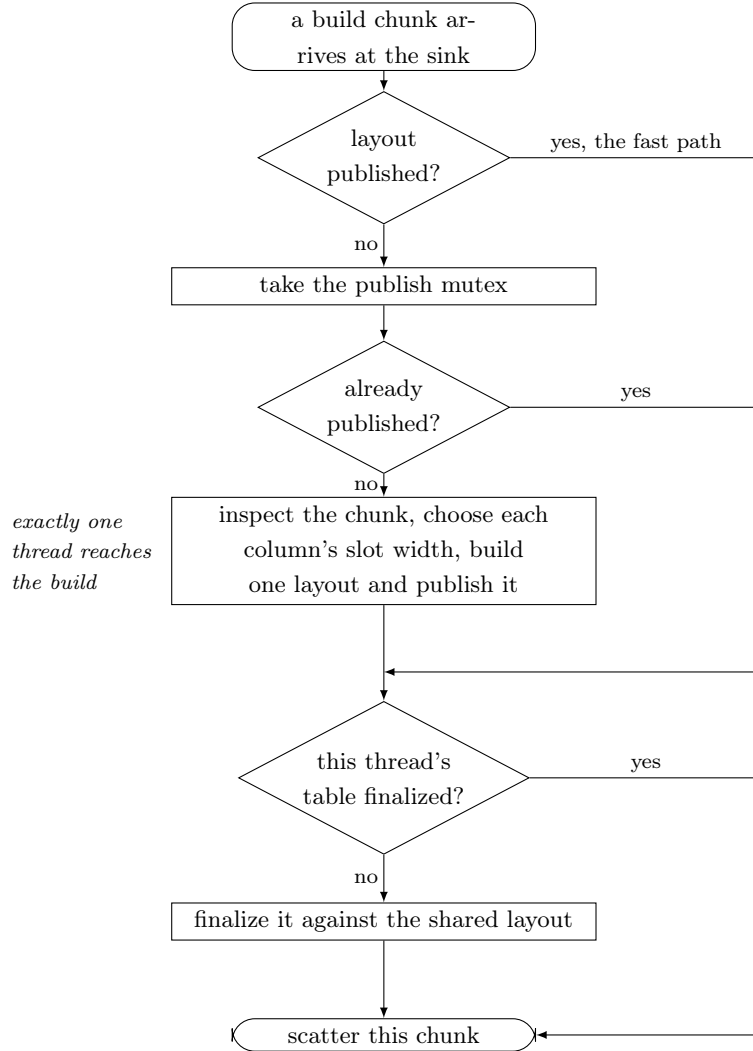


Figure 5.7: One build chunk through the layout gate. The outer check is for speed, since every chunk after the first takes it and never touches the mutex. The inner check is for correctness, since two threads can both see the flag unset before either acquires the lock, and only one of them may build the layout. Every thread past either check finalizes its own hash table against the same published layout, so all of them write rows in one byte format.

when the sink state is constructed and read as flags. Both follow from the plan alone, so neither can change once the query is running.

The caching operator's change is confined to how it seeds, appends to, and flushes its buffer. It seeds a dictionary column from the first chunk's entry, appends the indices of later chunks against that entry, and re-wraps the column on flush. This began as a fix for pipeline-global dictionaries alone, the ones this optimization depends on. It was later generalized to preserve any dictionary with a stable identifier, flattening a column when

its dictionary changes between chunks and leaving it alone when it does not.¹

5.8 Evaluation

This section measures the optimization in isolation, on micro-benchmarks built to exercise the join shape it targets, a first join that emits pipeline-global dictionaries, feeding a second join whose build side carries them. The first two run the same query, and differ only in one setting. The first measures the saving when the join runs in memory either way, in peak memory, operator time, and whole-query time. The second adds a memory limit, under which the same saving decides whether the join spills to disk at all. The third takes a different query and asks what the dictionary is worth to the operators above the join. Broader results on standard analytical workloads, and the effect of this optimization alongside the other contributions of this thesis, are reported in Chapter 7.

5.8.1 Setup and methodology

The first two experiments chain two of this thesis’s optimizations. A small dimension table is joined to a larger table on a `VARCHAR` key, which blocks the perfect hash join and forces the general hash-join path. That first join is small enough to trigger the emission of Chapter 4, so its payload columns leave it as pipeline-global dictionaries. The result becomes the build side of a second join against a large probe table, and it is on this second join that the optimization under test fires. The emission of Chapter 4 is active in both the enabled and the disabled condition, so the first join is identical across the two and the only thing that changes is whether the second join narrows its row store.

```

1  -- dim: 10 rows, ten inline VARCHAR payload columns, ten distinct each
2  -- mid: 5,000,000 rows, the build side of the second join
3  -- big: 10,000,000 rows, the probe side of the second join
4  SELECT COUNT(*), MIN(p1), MIN(p2), ..., MIN(p10)
5  FROM big
6  JOIN (SELECT mid.j2key, dim.p1, dim.p2, ..., dim.p10
7        FROM mid JOIN dim ON mid.dim_key = dim.dim_key) sub  -- first join
8  ON big.bkey = sub.j2key;                                     -- second join

```

Measurements were taken on an Apple M4 Pro with 24 GB of memory. Each query runs single-threaded after two warm-up runs, under DuckDB’s profiler, with join reordering disabled so the dimension stays on the build side.

¹<https://github.com/duckdb/duckdb/pull/23447>

5. THE DICTIONARY-SURVIVING HASH JOIN

Two further optimizer passes are disabled so the intended plan is exercised. `build_side_probe_side` picks a hash join’s build side from a width and cardinality heuristic, and with ten wide dictionary columns it places the dictionary-carrying relation on the probe side, where the second join never sees it. `compressed_materialization` inserts a projection that re-compresses the payload strings ahead of the second join, re-minting their dictionaries without the pipeline-global flag, so the sink declines them. Both are applied identically to the enabled and disabled conditions, so neither biases the comparison. The second is not independent of this optimization, though, since it compresses the same columns by another route, and whether it intercepts them depends on how many low-cardinality string columns the plan carries. These controls are specific to these micro-benchmarks, and the suite results of Chapter 7 disable no optimizer.

Every condition is run ten times with the optimization enabled and ten disabled, toggled by a single compile-time constant. The second hash join is the parent of the first in the plan, and its operator timing is read from its own node. Peak memory is the maximum the buffer pool’s allocation reaches over the query, reset at the start and read at the end; single-threaded, this value is deterministic. The sign convention throughout is that a reduction is positive, computed as the disabled figure minus the enabled figure over the disabled figure.

5.8.2 In-memory saving

The first experiment runs both joins in memory with no limit set, and measures what the optimization saves. A ten-row dimension carries ten inline `VARCHAR` payload columns of ten values each. It joins to a five-million-row table, which becomes the build side of a second join against a ten-million-row probe. The emission of Chapter 4 turns the ten payload columns into pipeline-global dictionaries of ten entries, so the second join narrows each sixteen-byte string slot to a one-byte index. All strings are short enough to store inline, so no string heap is involved on either side, and the measurement is a pure row-store layout effect with nothing else to attribute it to.

The narrowing removes ten columns of fifteen bytes each from five million build rows, a predicted 750 megabytes of row store. Table 5.1 reports the three measurements.

Peak memory is the headline. It falls by 750.5 megabytes, which matches the 750 megabytes the layout was predicted to save to within a tenth of a percent. The whole-query peak is set when the second join’s row store is fully built and pinned, so removing the string slots from that row store moves that peak down by exactly their size. With all strings inline there is no heap term on either side to muddy the attribution, so the change in peak memory is the change in the row-store layout.

5.8 Evaluation

metric	disabled	enabled	reduction	improvement
peak memory (MB)	1179.4	428.8	63.6%	$2.75\times$
second join (ms)	524.8	315.5	39.9%	$1.66\times$
whole query (ms)	1040.3	796.8	23.4%	$1.31\times$

Table 5.1: The in-memory saving, at a five-million-row build side with ten columns narrowed to one-byte indices. Peak memory is deterministic; the timings are geometric means over ten runs.

The second join runs 39.9 percent faster, a 209-millisecond saving on the operator. Narrowing the slot is not free-standing bookkeeping. The build-side scatter writes a one-byte index per column rather than copying a sixteen-byte string header, and the probe-side gather re-emits a dictionary vector over the pinned copy rather than gathering ten million rows of ten string values. The same copies that shrink the row store are the copies the operator no longer makes, so the memory saving and the operator speedup are two views of one mechanism.

The whole query is 23.4 percent faster. The optimized join is one operator on the critical path, and the downstream aggregate and the table scans it does not touch make up the rest of the time. The aggregate also reads the surviving dictionary back through its index rather than exploiting it, so the compression is undone as soon as the join hands its output on. The operator result is the one that isolates what the optimization does, and there the saving is largest after peak memory.

5.8.3 Avoiding a spill

The second experiment places the same query under a memory limit, where the row-store saving stops being a matter of degree and decides the execution strategy. The query, the tables, and the plan are exactly those of the previous experiment, a ten-column build side of five million rows, narrowed to one-byte indices. The only addition is a memory limit of 600 megabytes. Without the optimization the second join’s build side is about 1171 megabytes, and the engine grants it a reservation of about 270 megabytes, so the join spills to disk and runs as an external hash join. With the optimization the ten string slots narrow to one-byte indices, the build side falls to about 421 megabytes, and it fits in memory.

Avoiding the spill makes the whole query 1.84 times faster and the second join 3.12 times faster. The larger operator ratio is expected, since the spill cost falls entirely inside that

5. THE DICTIONARY-SURVIVING HASH JOIN

metric	disabled (spills)	enabled (in memory)	reduction	speedup
second join (ms)	1032.59	330.57	68.0%	3.12×
whole query (ms)	1541.13	837.48	45.7%	1.84×

Table 5.2: Avoiding a spill, at a memory limit of 600 megabytes, on the query of Table 5.1. The build side is about 1171 megabytes without the optimization and about 421 with it. Geometric means over ten runs.

one join, while the whole-query figure also carries the unchanged cost of the ten-million-row scan and the rest of the plan. Decomposing a representative run confirms it. The second join accounts for 708 of the 711 milliseconds of whole-query difference, and every other operator changes by a small amount. The spill cost lands inside the second join’s own timing, since partitioning the build to disk and probing it back in rounds are part of that operator’s work, so the operator timing is a faithful measure of the spill.

Running the same query with and without a limit separates the two benefits. With no limit the optimized join saves about 243 milliseconds, from the cheaper scatter and gather over the narrowed row store. The 600-megabyte limit adds another 461 milliseconds, the external cost that only the spilling run pays. So about two thirds of the speedup here is the avoided spill, and one third is the in-memory compression already measured above. Between the two experiments the enabled run gains about 40 milliseconds of bookkeeping under the limit, since its 421-megabyte build fits inside the budget. The entire extra win comes from the disabled run going to disk.

5.8.4 Downstream impact

The two experiments above measure the join. This one asks what the dictionary is worth after it. A dictionary that survives a join leaves it under the identifier it arrived with, so any operator above that keys a cache on that identifier can use it without being changed. The experiment uses the example query of Figure 5.1.

The query totals revenue by country over three tables. A ten-row `country` table gives each country a name, `customer` assigns 1,500,000 customers to those countries, and `orders` holds 15,000,000 orders. The country name is stored once per country and has to reach every order row before the query can group on it.

```

1 SELECT n.country_name, COUNT(*) AS n_orders, SUM(o.total) AS revenue
2 FROM   country  n
3 JOIN   customer c ON c.country_code = n.country_code
4 JOIN   orders   o ON o.customer_id  = c.customer_id
5 GROUP BY n.country_name;

```

DuckDB plans this as two joins. The first builds on the ten-row `country` table, which activates the emission of Chapter 4 and produces `country_name` as a pipeline-global dictionary of ten entries. That join’s output is the build side of the second. There the optimization under test narrows the column from a sixteen-byte string slot to a one-byte index. Above the second join sit a projection that compresses the column and a hash aggregate that groups on it. Neither was modified for this work.

Setup follows Section 5.8.1 with one difference. No optimizer pass is disabled here, so the plan measured is the one DuckDB produces on its own. Two binaries were built, one with both optimizations off and one with both on. Each query ran ten times per binary after two warm-up runs, the two arms interleaved within each repetition. A control query holding no hash join measured the noise floor at about 0.4 percent.

operator	disabled	enabled	reduction	speedup
the emitting join	4.45	4.08	8.3%	1.09×
the dictionary-surviving join	304.19	245.49	19.3%	1.24×
projection	25.50	0.99	96.1%	25.70×
hash aggregate	72.51	39.66	45.3%	1.83×
whole query	487.69	368.46	24.5%	1.32×

Table 5.3: Wall time in milliseconds for each operator of the plan, read bottom to top, and for the query as a whole. The emitting join’s figure is its full operator cost, its timing plus the measured dictionary build. Geometric means over ten runs.

The whole query runs 1.32 times faster. Where that saving falls is the result. The projection and the hash aggregate together save 57.35 milliseconds, nearly as much as the optimized join’s own 58.70. The aggregate resolves ten groups once rather than hashing 15,000,000 keys, and the projection evaluates its expression over ten dictionary entries rather than 15,000,000 rows. Neither operator knows about this optimization. Both improve because the column reaching them is still a dictionary under an identifier they have already seen.

5. THE DICTIONARY-SURVIVING HASH JOIN

6

Dictionary-Aware Probing

The first contribution generates pipeline-global dictionaries, and the second keeps them alive through the join's build side. This chapter aims to exploit them on the probe side. It applies whenever the join key reaches the probe dictionary-encoded, whether storage produced that dictionary for a low-cardinality column or an upstream operator emitted it as one of the earlier chapters described. However the dictionary arrives, the ordinary probe discards its structure. For each row it hashes the key, looks up the head of the matching chain in the hash table, and re-checks the key bytes against the build row. On a probe side of a million rows that is a million of each, for a key that might take only twenty-five values. Every row with the same key gets the same result, so those twenty-five lookups are repeated over and over. This chapter resolves each distinct key once and reuses the result for every row that shares it.

The chapter proceeds from the design to its measurement. Section 6.1 presents the approach, including the constant-key case, and Section 6.2 follows one probe through it. Section 6.3 then gives the conditions under which the fast path runs, Section 6.4 records where the change sits in the code, and Section 6.5 measures the result in isolation.

6.1 The approach

The aggregate operator already resolves its grouping keys this way. When a low-cardinality dictionary column arrives, it looks up each distinct value once and fans the result out to the rows. This chapter brings the same principle to the probe side of a hash join. The probe has its own machinery to fit into, though. It walks chains, and it has to get null semantics right.

6. DICTIONARY-AWARE PROBING

The join can reuse a resolved result across chunks as long as they carry the same dictionary identifier, since the same identifier means the same values indexed the same way. How long an identifier lasts depends on where the dictionary came from. A storage dictionary is fixed within one row group but changes at the next, so its identifier rotates as the scan advances and the reuse resets at each boundary. A pipeline-global dictionary produced by one of the earlier chapters keeps a single identifier for the whole operator, never swapping the dictionary, so its results can be reused for the entire scan.

Instead of resolving each row, the join resolves each dictionary slot once. Resolving a key means finding its head pointer, the start of the hash-table chain that holds every build row the key might match. The join keeps a small cache, one head pointer per slot, tied to the dictionary's identifier. What a chunk costs then depends on how much of the dictionary the cache already holds. We call the cache cold when it holds no resolved pointers for the current dictionary and warm once earlier chunks have filled some in. Both terms describe what the join's per-identifier cache holds. Neither is about the processor's cache.

On the first chunk of a dictionary, the cache is cold. The join makes two passes over the chunk. The first pass reads each row's slot and collects the set of slots that appear, then resolves that set against the hash table in one batch, storing each slot's head pointer in the cache. The second pass reads each row's slot again and copies the cached head pointer for that slot into the probe's working state, so every row that shares a slot receives the same pointer. A dictionary of twenty-five values costs at most twenty-five resolutions, however many rows the chunk holds.

On a later chunk carrying the same identifier, the cache is warm. The join keeps whatever it resolved on earlier chunks, so it only has to resolve slots it has not seen before. It walks the chunk once to find any slot appearing for the first time, resolves just those against the hash table, and adds them to the cache. Every slot already in the cache is reused with no lookup. A chunk whose slots were all seen earlier does no hash-table work at all; it only copies cached pointers into place.

The cache fills up as the scan proceeds. Once every slot in the dictionary has been resolved, the cache is saturated, and no later chunk can bring a slot that is not already there. At that point the join stops walking chunks to look for new slots and goes straight to the fan-out, copying a cached pointer for each row and nothing more. On a long scan most chunks run in this saturated state, so the cost per chunk settles to a single pass that copies pointers, with no hashing and no lookups. The resolution work is paid once for the whole dictionary rather than once per chunk or once per row.

The aggregate operator did the same redundant work. Its dictionary path walked every chunk looking for new slots, even once its cache was complete, so every walk after that found nothing. We implemented the same mechanism on the aggregate path, and both operators now stop walking once the dictionary is exhausted.¹

The cache changes how a pointer is found, never what the probe does with it, so correctness comes from machinery that was already there. This holds for chains, for null keys, and for a change of identifier.

A build side may hold several rows with the same key, chained together behind one hash-table entry. The cache does not interfere. A hash-table lookup returns the head of the chain, so the cache stores a chain head. Fanning that head out to every row of the slot is correct, since each of those rows would have resolved to the same head under the ordinary path. The chain walk that follows runs per row, exactly as before. Resolving the head once and reusing it changes only the lookup, never the walk.

Null keys are handled by the same filter the ordinary probe uses. Under an equality join a null probe key matches nothing, so the fan-out drops every row whose key is null before any pointer is written. Under an `IS NOT DISTINCT FROM` join a null matches a null on the build side, so those rows are kept and the hash-table's own row matcher resolves the null slot correctly. The cache path routes both through the existing null filter, so its output matches the ordinary path row for row.

A dictionary does not last forever. When the scan crosses into a new row group, the column arrives with a fresh identifier, and the cache built for the old one no longer applies. The join notices the identifier has changed, discards the old cache, and binds a new one to the new identifier. The next chunk is cold again, and the cycle repeats. Between rotations the cache stays warm.

The fast path runs only when the probe key arrives as a dictionary vector. When it does not, the join runs its ordinary per-row probe instead, exactly as before. Even when the key is a dictionary, the fast path checks that caching will pay off and steps aside if it will not; Section 6.3 gives the exact conditions. Whichever path runs, the matches are the same. The fast path changes how the pointers are found, not what the join returns.

6.1.1 Constant keys

A constant vector is a dictionary with a single value, shared by every row in the chunk. It arises in two ways. Storage constant-compresses a segment whose values are all identi-

¹<https://github.com/duckdb/duckdb/pull/22813>

6. DICTIONARY-AWARE PROBING

cal, though only for numeric and boolean columns, never for strings, which it dictionary-compresses instead. An operator can also emit one, as when a literal is projected into a column and carried down to the join.

With one distinct value there is nothing to cache and no identifier to key a cache on, so the whole per-slot mechanism collapses. The join resolves the single value once, then fans its head pointer out to every row of the chunk. Each chunk pays exactly one lookup. There is no warm regime, since there is nothing to carry across chunks, but one lookup for a chunk of two thousand rows is already negligible. Nulls need no special handling here either. The single key is resolved through the same row matcher, and the fan-out applies the same null filter as every other path.

A chunk arrives as one shape or the other, so only one path ever runs. The constant path is tried first. It never touches the dictionary cache, so a column whose chunks alternate between the two shapes stays safe, since a constant chunk cannot overwrite the resolved slots a dictionary chunk under the same identifier still depends on.

6.2 A worked example

A `sales` table is joined to a small `country` table on the country code, and the `sales` side is the probe. Storage hands the join its key column one chunk at a time, and each chunk arrives as a dictionary vector. Take a dictionary of five country codes and a chunk of eight rows. The chunk carries three things, shown in Figure 6.1. A child holds the five values, a selection vector gives each row the slot it reads from that child, and an identifier names the dictionary. Only the child and the selection vector are physically stored, so eight rows cost eight slot numbers rather than eight strings.

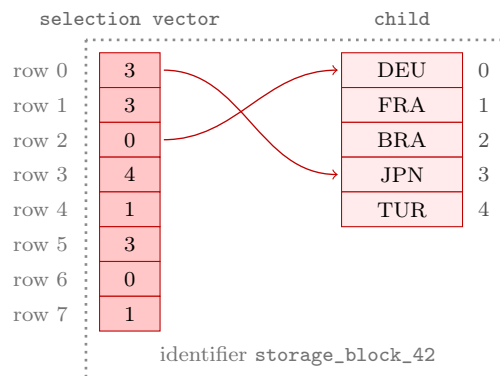


Figure 6.1: The incoming chunk. Row 0 reads slot 3, so its key is JPN, and row 2 reads slot 0, so its key is DEU. The eight rows reference only four of the five slots.

6.2 A worked example

The cache has not seen this identifier, so it is cold. The join makes one walk over the selection vector to collect the slots that appear, in the order they are first seen. Rows 0 and 1 both read slot 3, which is new, so slot 3 is recorded once. Row 2 brings slot 0, row 3 brings slot 4, and row 4 brings slot 1. Every row after that repeats a slot already seen. The walk ends with four slots to resolve, $\{3, 0, 4, 1\}$, and slot 2 never appears in this chunk at all.

The join then slices those four values out of the child and probes the hash table once, on four keys rather than eight rows. Each probe returns a head pointer, which the join stores in the cache at that value's slot. Figure 6.2 shows the result.

slot	child	cache
0	DEU	head pointer
1	FRA	head pointer
2	BRA	<i>not resolved</i>
3	JPN	head pointer
4	TUR	head pointer

Figure 6.2: The per-slot cache after the chunk. The four slots the chunk referenced hold a resolved head pointer. Slot 2 was never referenced, so it stays empty.

A second walk fans the cache out to the rows. For each row the join reads its slot from the selection vector and copies that slot's cached head pointer into the probe's working state. Rows 0, 1, and 5 all read slot 3 and receive the same pointer, resolved once and reused three times. From here the probe proceeds exactly as it always does. It walks each row's chain and emits the matches, unaware that the head pointers came from a cache rather than from eight separate lookups.

The next chunk of the same row group carries the same identifier, so the cache survives. The join has resolved four of the five slots, though, and it cannot know whether the fifth appears in this chunk without looking. It walks the selection vector again. Suppose this chunk does reference slot 2. The walk finds it, the join probes the hash table for that one value, and stores its head pointer in the cache. Every other row reads a slot already resolved, so this chunk costs a single lookup for eight rows.

Now all five slots hold a head pointer. The join tracks how many it has resolved, so it knows the dictionary is exhausted, and no chunk can bring a slot it has not already seen. For every chunk after this one, the walk is skipped entirely. All that is left is to read each row's slot and copy the cached pointer, with no hashing and no hash-table lookups at all. When the scan crosses into a new row group the identifier changes, the cache is discarded, and the next chunk starts cold.

6.3 Activation

The fast path is guarded in three layers, from a once-per-operator check down to a per-chunk check inside the probe itself. Each layer is cheap enough to run at its frequency, and the earlier layers exist so the later ones run only when they can succeed.

The first layer is decided once, when the operator's state is built. The join must be a plain equality or `IS NOT DISTINCT FROM` join on a single key, and that key must be a column reference rather than a computed expression, since only a scanned column arrives dictionary-encoded. The key must not be a struct, which storage does not dictionary-encode. Two whole-operator conditions also exclude the fast path. An external join re-partitions the build side mid-probe, which would leave the cache pointing at stale build rows, and a perfect hash join uses a different probe path entirely. Both are handled by the ordinary path.

The second layer is decided once per chunk, before the fast path is entered. The key vector must be a dictionary vector.

The third layer is decided inside the fast path itself, and it is where the fast path is finally accepted or rejected. It re-checks that the key really is a dictionary vector, then works out whether resolving per slot will be cheaper than resolving per row. That depends on whether the dictionary carries an identifier, since the identifier is what lets the cache survive from one chunk to the next.

A dictionary with an identifier can be cached across chunks, so its cost is paid once and amortized over the whole scan. The join accepts it as long as it holds fewer than twenty thousand values. Past that the per-slot array grows large enough that walking and allocating it costs more than the lookups it saves, however many chunks share it.

Without an identifier, nothing carries to the next chunk, so the work has to pay for itself inside the one chunk it appears in. Resolving a dictionary of D values costs D lookups and saves the join from doing N , so it only helps when D is well under N . The join requires the dictionary to hold fewer than half as many values as the chunk has rows, which on a full chunk of two thousand and forty-eight rows means fewer than one thousand and twenty-four.

If any of these checks fails, the fast path returns without doing anything, and the caller runs the ordinary probe.

6.4 Implementation

The optimization adds one fast path for a dictionary key and one for a constant key, beside the ordinary per-row probe.¹ Each probe chunk takes exactly one of the three. The two fast paths leave the probe in exactly the state the ordinary path would, so everything downstream of them, the chain walk and the match emission, runs without knowing which one filled it. The change is small. It adds 306 lines across three engine files and removes none, excluding tests and benchmarks.

The cache is per thread and lives on the join’s probe state, holding the identifier it is bound to, one head pointer per dictionary slot, and a count of the slots resolved since the last rebind. The hash table’s chain walker and row layout are untouched.

The fast path reuses the function the ordinary probe already uses to turn a batch of keys into chain heads. It calls that function on the distinct slots rather than on every row, which is the whole saving.

6.5 Evaluation

This section measures the optimization on a query that chains it to the small build-side dictionary emission of Chapter 4. The join key it probes on is not scanned from storage. It is manufactured mid-pipeline by that emission, which makes the query a direct test of whether these contributions compose. Broader results on standard analytical workloads are reported in Chapter 7.

6.5.1 Setup and methodology

The experiment chains this optimization to the emission of Chapter 4. A fifty-million-row fact table is joined to a nineteen-thousand-row dimension on a `VARCHAR` key, which triggers the emission and hands the dimension’s `code` column downstream as a pipeline-global dictionary. That column is the probe key of a second join, against a four-million-row build side, and it is on this second join that the optimization under test fires.

```

1  -- dim:  19,000 rows, the build side of the first join
2  -- dim2: 4,000,000 rows, the build side of the second join
3  -- fact: 50,000,000 rows, the probe side, 19,000 distinct keys
4  SELECT COUNT(*), COUNT(d2.label), MIN(d2.label), MAX(d2.label)
5  FROM fact f
6  JOIN dim d ON f.dim_key = d.dim_key    -- first join, emission fires
7  JOIN dim2 d2 ON d.code = d2.code;      -- second join, probing fires

```

¹<https://github.com/duckdb/duckdb/pull/22789>

6. DICTIONARY-AWARE PROBING

The build side of the second join is deliberately large. Its four million rows build a hash table of about 336 megabytes, far larger than the processor’s cache, and the probe references nineteen thousand distinct keys scattered across it. This is a more realistic setting than a join against a handful of rows, and it is one where a lookup into the hash table is a cache miss rather than a cache hit. Only nineteen thousand of the four million build keys are ever matched, so the table stays large while the probe resolves to a bounded set of keys.

Measurements were taken on an Apple M4 Pro, single-threaded, with join reordering disabled so the optimizer does not join the two dimensions against each other first and break the chain. The first join is a control, since the optimization never fires on it. Its probe key arrives flat rather than as a dictionary. The two conditions are built as separate binaries and run interleaved within each repetition, so any drift in the machine’s speed over the session falls on both equally. Two warm-up repetitions are discarded and ten are timed. A reduction is positive, the disabled figure minus the enabled figure over the disabled figure.

6.5.2 Probing a pipeline-global dictionary

With the hash table this large, the optimization more than doubles the speed of the second join and cuts about a sixth from the whole query. The mechanism is the one from the rest of this chapter, at a scale where the lookups it removes are expensive. The probe carries fifty million rows over nineteen thousand distinct keys. Without the optimization the join looks up every row. With it, each key is resolved once. The first chunk finds the cache cold and resolves the keys it references, and each warm chunk after it resolves any key it is the first to see. With nineteen thousand keys the cache keeps filling for ten chunks, then saturates, and no later chunk looks anything up. In total the query resolves nineteen thousand keys in place of fifty million per-row lookups.

Table 6.2 reports the timings. The second join runs more than twice as fast, a reduction of 53.6 percent, and the whole query 1.21 times faster. The saving is larger than a small dimension table would show, since each lookup the optimization removes is a cache miss.

The whole-query reduction is smaller than the join’s. The optimization removes the hash-table lookups on the second join and nothing else. That join still walks each match chain and gathers the payload for every one of the fifty million rows it emits, and the

	disabled	enabled
hash-table lookups	50,000,000	19,000
chunks with lookups	24,415	10

Table 6.1: Hash-table work across the whole query, over 24,415 probe chunks. Without the optimization every probe row is looked up. With it, nineteen thousand keys are resolved over the first ten chunks and reused for the rest of the scan.

metric	disabled	enabled	reduction	speedup
second join (ms)	446.63	207.23	53.6%	2.16×
whole query (ms)	1565.24	1295.52	17.2%	1.21×

Table 6.2: Geometric means over ten interleaved runs.

fact-table scan, the first join, and the aggregate over every joined row are untouched. The lookups are one part of the query’s work, so removing them cuts the whole query by less than it cuts the join.

The dictionary the second join probes carries one identifier for the whole scan, so the cache is built once and reused across all fifty million rows. A dictionary scanned from storage would rotate its identifier at every row group, rebuilding the cache at each boundary. The dictionary here is emitted by the first join rather than read from storage, which is what lets a single binding cover the entire scan.

6. DICTIONARY-AWARE PROBING

7

Evaluation

The three preceding chapters each measured one contribution in isolation, on micro-benchmarks built to exercise the join shape it targets. This chapter asks a different question. It runs all three together, on standard analytical workloads that were not designed with them in mind, and reports what happens.

7.1 Methodology

All measurements were taken on an Apple M4 Pro with 24 GB of memory, single-threaded, with DuckDB’s optimizer fully enabled and no plan hints. The earlier chapters disabled optimizer passes to hold a plan in place. Here the plans are the ones DuckDB chooses on its own.

Each contribution is guarded by a compile-time flag, so the experiment compares two separately compiled binaries rather than one binary with a runtime switch. Setting a flag to false makes its activation check fail every time, so the disabled binary never enters the new code path. With all three flags off, no activation of any kind was observed across the instrumented runs, so the disabled binary is a clean baseline.

Timings come from DuckDB’s profiler, which records a wall-clock time for every operator node in the plan. A hash join’s operator time is read from its own node, so a query’s gain can be traced to the individual joins that produced it rather than inferred from the whole-query figure. The small build-side dictionary emission of Chapter 4 needs one addition. It builds its dictionaries in a separate pass after the build side is finalized, which falls outside the node’s timed execution, so the profiler would not count it. That pass is therefore timed and added into the join’s reported operator time, which keeps the emission from looking cheaper than it is.

7. EVALUATION

Peak memory needs its own instrument, since DuckDB tracks how much memory it is using at any moment but never records the maximum that figure reaches over a query. That maximum is the quantity of interest. A counter was therefore added at the buffer pool’s memory-usage update, the single point every allocation passes through. It takes a new value whenever the running total exceeds the peak recorded so far. The counter is reset at the start of the query and read at the end, with a granularity of two megabytes. Run single-threaded, it reproduces exactly on TPC-H and TPC-DS. On IMDB the peak varies from run to run, so the memory figures below are geometric means over the ten runs.

Every query runs ten times against each binary after two warm-up runs, and the two alternate back to back within each repetition. Any drift in CPU frequency, thermal state, or page cache is therefore shared by both runs and cancels in the ratio. Figures are geometric means over the ten runs. Even a query that activates none of the optimizations varies by one percent between the two binaries. That spread is the noise floor a real change has to clear.

Attribution of a pipeline-global dictionary to its producer is direct. Both producers, the emission and the perfect hash join, tag each dictionary they create with a unique identifier. The point where the surviving join captures a dictionary looks that identifier up to name the producer, so the share each supplies is counted rather than inferred from the plan shape.

7.2 IMDB

7.2.1 The benchmark

The Join Order Benchmark runs 113 queries over the IMDB dataset, a real-world schema of films, people, companies, and the relationships between them [37]. It was built to stress join ordering, so its queries are deep, with as many as sixteen joins each, and its data carries the correlations and non-uniform distributions a generated data set does not [37]. Titles, person names, and free-text notes are strings, most of them long. Some of its tables hold tens of rows and others tens of millions. The values a join carries out of a small one are therefore repeated across the rows of a large one. Small tables of this kind are not particular to this benchmark. Most tables in a cloud warehouse fleet hold under a million rows [47], and in a survey of business intelligence datasets the small ones serve as dimension tables that map codes to names [55].

That shape is what the contributions of this thesis need. The emission needs a build side small enough to turn into a dictionary. The surviving join needs payload strings long

enough that the row store would otherwise spend real time copying them. The probing optimization needs join keys that arrive dictionary-encoded. IMDB supplies all three, though the third far less often than the other two. The results below measure the contributions on a workload that produces their target shape without having been designed to.

7.2.2 Activation

All three optimizations activate, and they activate widely. Table 7.1 reports where.

optimization	queries	sites
small build-side dictionary emission	94 of 113	175 joins
dictionary-surviving join	60 of 113	193 columns
dictionary-aware probing	41 of 113	53 joins

Table 7.1: Activation across the 113 queries of the Join Order Benchmark. Only three queries activate none of the three optimizations.

Ninety-four of the 113 queries emit at least one build-side dictionary, sixty narrow at least one column by keeping its dictionary alive through a join instead of materializing the values, and forty-one run at least one compressed probe. Only three queries activate nothing at all, so the shapes these optimizations target are common in the benchmark.

The reason is the schema. IMDB has small lookup tables for role, information, company, and link types, joined to large fact tables like `cast_info` and `movie_info`. That pairing is exactly the shape that emits a dictionary on the build side and probes millions of rows against it. The dictionaries the emission produces are then available for the surviving join to capture downstream, and it draws at least one from the emission in 43 of the 60 queries where it activates.

7.2.3 Results

Table 7.2 reports the ten queries that gain most, with their peak memory alongside.

The distribution is concentrated rather than broad. Query 7c runs 2.23 times faster and its peak memory falls by 91 percent, from 4311 to 385 megabytes. The three queries of the 13 family, 13b, 13c and 13d, gain between 6 and 11 percent. Below that the gains thin out quickly, and most of the 110 activated queries land within a couple of percent of where they started. The suite as a whole moves by 1.4 percent.

That shape is worth stating plainly. These optimizations do not make a benchmark uniformly faster. They find the joins whose cost is concentrated in copying values that did

7. EVALUATION

query	active	wall time (ms)				peak memory (MB)		
		disabled	enabled	red.	speedup	disabled	enabled	red.
7c	1, 2	490.7	220.5	55.1%	2.23×	4311.1	384.6	91.1%
13c	2	106.9	95.4	10.8%	1.12×	157.8	105.1	33.4%
13b	2	120.1	108.1	9.9%	1.11×	160.4	108.2	32.5%
13d	2	99.0	92.7	6.4%	1.07×	99.1	72.7	26.6%
23a	1, 2	102.3	96.8	5.4%	1.06×	80.1	80.0	0.1%
16c	1, 2	192.1	185.5	3.4%	1.04×	302.3	287.0	5.1%
16d	1, 2	182.5	178.5	2.2%	1.02×	292.1	278.3	4.7%
1b	1	9.2	9.0	2.1%	1.02×	16.5	16.5	0.0%
11a	1	11.9	11.7	2.0%	1.02×	19.4	19.4	0.0%
18b	1, 2, 3	215.6	211.9	1.7%	1.02×	345.7	345.2	0.2%

Table 7.2: The ten queries with the largest end-to-end gain. The *active* column names which contributions activate, where 1 is the small build-side dictionary emission, 2 the dictionary-surviving join and 3 dictionary-aware probing. The two queries that activate only the emission do not narrow a row store, so their peak memory is unchanged.

not need copying, and on those joins the gain is large. The sections that follow take each contribution in turn and show where its gain comes from.

7.2.4 Small build-side dictionary emission

The emission of Chapter 4 activates on more joins than either of the others, 175 across 94 queries. It builds its dictionaries once per join, at finalize. Its cost is therefore fixed per join, while its work grows with the number of rows the join emits. Table 7.3 reports the three joins where it saves most operator time.

query	rows emitted	dict build (ms)	operator (ms)		reduction	
			disabled	enabled	operator	query
22c	2,610,972	0.21	12.35	9.82	20.5%	0.4%
28c	1,151,351	0.06	4.65	3.91	15.8%	1.2%
7c	1,618,081	0.05	10.84	9.24	14.8%	55.1%

Table 7.3: The three hash joins where the emission saves most operator time. Each emits over a million rows for a dictionary build costing a quarter of a millisecond or less, paid once and folded into the *enabled* figure. The operator gain reaches the whole query only for 7c, whose dictionaries feed the dictionary-surviving join; 22c and 28c have no consumer downstream.

On each of these joins the dictionary build costs a quarter of a millisecond or less, paid once, while the join emits over a million rows and the operator runs 15 to 20 percent faster. On its own the operator saving adds up to only a small percentage of the whole

query. The real benefit comes when a downstream operator keeps the dictionary rather than materializing it back into values. The query-reduction column shows the split. 7c, whose dictionaries feed the dictionary-surviving join, runs 2.23 times faster end to end, while 22c and 28c, with no consumer, move under two percent.

Most of the dictionaries the surviving join narrows come from the small build-side dictionary emission, across the whole benchmark. 7c is an instance of that arrangement rather than an exception. Table 7.4 reports which operator produced each pipeline-global dictionary the dictionary-surviving join went on to capture.

producer	queries	columns captured
small build-side emission	39 exclusively	141
perfect hash join	17 exclusively	52
both	4	
total	60	193

Table 7.4: Where the dictionary-surviving join’s pipeline-global dictionaries come from. The emission of Chapter 4 supplies 141 of 193 columns.

The emission produces 141 of the 193 columns the next optimization narrows, and 39 of the 60 queries that narrow a column are fed by it alone. Query 7c, the largest result in this benchmark, draws all four of its narrowed columns from general hash joins running the small build-side dictionary emission, and none from a perfect hash join. Turn the emission off and the sink has no pipeline-global dictionary to capture there, and 7c’s 2.23 times speedup and its 3.9 gigabytes of saved memory go with it.

On this workload the small build-side dictionary emission is the supplier the surviving join depends on, and the two have to be judged together. Measured alone it looks small. Without it, the benchmark’s largest result has nothing to work on.

7.2.5 The dictionary-surviving join

The dictionary-surviving join of Chapter 5 activates on 143 joins across 60 queries, and every large win in the benchmark comes from it. Table 7.5 reports its five best.

The two joins of query 7c lead the table, each cutting its operator time by more than 92 percent, and together they carry the whole query to a 2.23 times speedup. The 13 family follows, with operator reductions between 32 and 50 percent.

The operator saving shows up directly as memory, which is the metric the surviving join was built to move. Table 7.6 reports peak memory on the queries where it falls most.

7. EVALUATION

query	join	build rows	operator (ms)		reduction	
			disabled	enabled	operator	query
7c	linked_movie_id = movie_id	1,618,081	249.26	18.54	92.6%	55.1%
7c	id = movie_id	109,842	32.59	2.54	92.2%	55.1%
13b	id = movie_id	1,153,798	26.86	13.58	49.5%	9.9%
13c	id = movie_id	1,153,798	26.87	13.65	49.2%	10.8%
13d	movie_id = movie_id	552,328	18.48	12.48	32.5%	6.4%

Table 7.5: The five hash joins where the dictionary-surviving join saves most, in operator time. Each is identified by its join keys. The last column is the reduction in the whole query’s wall time, which the two joins of 7c share between them.

query	disabled (MB)	enabled (MB)	saved (MB)	reduction
7c	4311.1	384.6	3926.5	91.1%
13c	157.8	105.1	52.8	33.4%
13b	160.4	108.2	52.1	32.5%
13d	99.1	72.7	26.4	26.6%
16c	302.3	287.0	15.3	5.1%
all 111 queries	30714	26596	4118	13.4%

Table 7.6: Peak memory, geometric means over ten runs.

Query 7c’s peak falls by 3926 megabytes, which is 91 percent of what it was. Across the suite peak memory falls by 4118 megabytes, or 13.4 percent. Almost all of that is this one query, so it is worth following its plan to see where 3.9 gigabytes went.

Figure 7.1 reproduces the upper part of the physical plan, from the emitting join up, with each join’s build edge [B] and probe edge [P] and its actual cardinalities.

The surviving join can only narrow a column that already arrives as a pipeline-global dictionary, so that dictionary has to come from a producer upstream. For both HJ5 and HJ7 in Figure 7.1, the producer is HJ4. That join has a build side of 12,595 rows carrying two payload columns, a person’s name and a free-text biography, against `cast_info`. The optimizer estimates that probe at roughly 36 million rows, though the join processes only 616,562. That estimate clears the ratio threshold, so the small build-side dictionary emission of Chapter 4 activates at HJ4 and creates two dictionaries of 12,595 entries, one for each payload column.

HJ4 emits 1,618,081 rows, and every output chunk carries those two dictionaries. Its output is HJ5’s build side, and the surviving join captures both dictionaries there, narrowing each 16-byte string slot to a 2-byte index. This is the join that wins the 3.54 gigabytes,

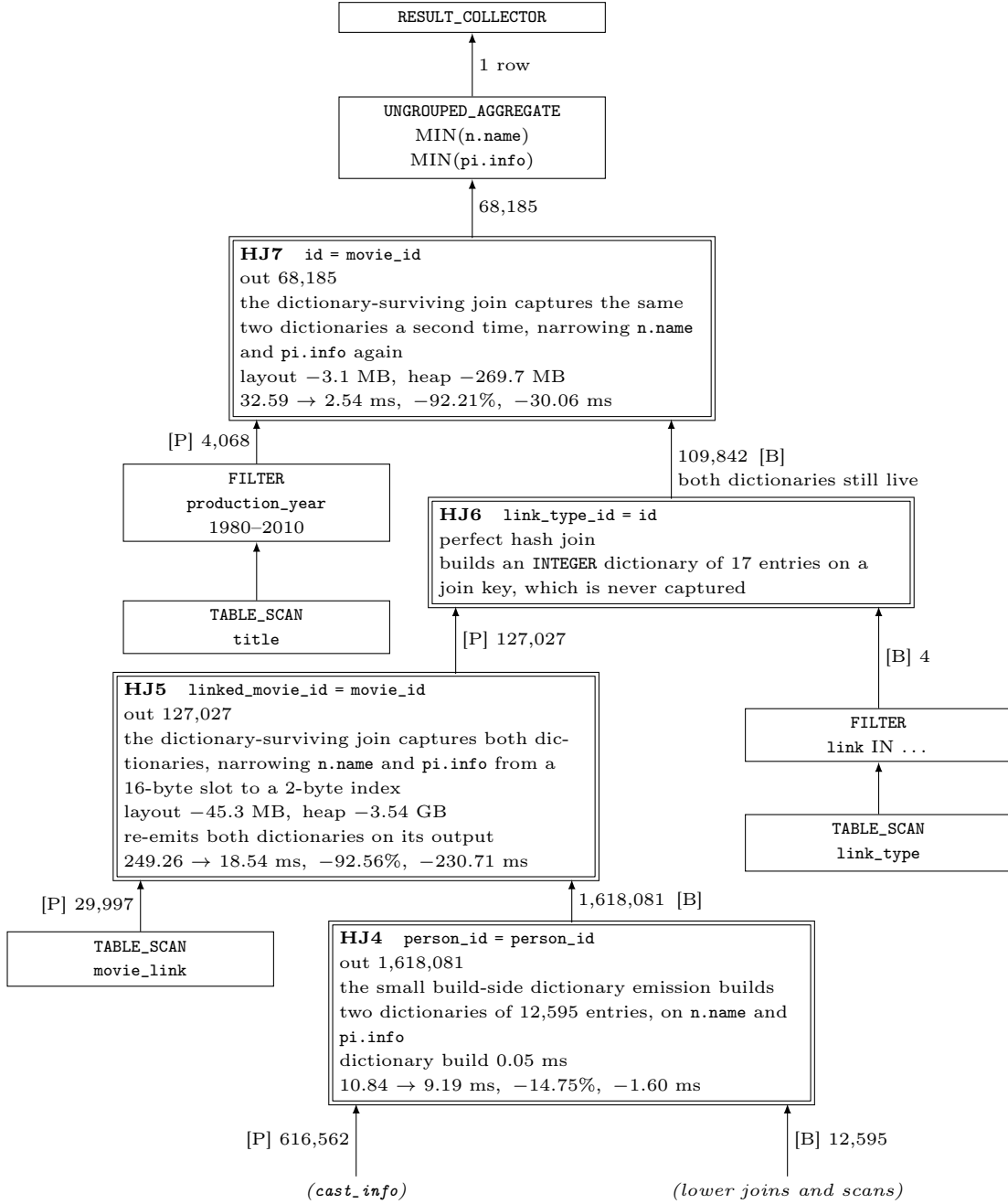


Figure 7.1: The upper part of the physical plan of query 7c, read bottom to top. [B] marks each join’s build edge and [P] its probe edge, with actual cardinalities. Every runtime and memory figure is a change, so a negative value is a saving.

7. EVALUATION

because one of those columns holds biography prose averaging over 2,000 bytes a value, copied once for each of its 1.6 million build rows on the native path. Rather than materialize the strings back on the way out, HJ5 re-emits the dictionaries on its own output, so they stay live past the join that consumed them.

HJ5’s output is the probe side of HJ6, the join above. HJ6 is a perfect hash join, and it builds a dictionary of its own, on the integer join key it builds on. It never touches the two payload columns though, which are probe-side here, so HJ6 passes them straight up and preserves both dictionaries untouched on its output. HJ6’s own dictionary sits on a build-side join key that no later sink builds on, so the surviving join leaves it alone. The plan therefore shows the two cases together, the payload dictionaries that survive and the key dictionary that is ignored.

HJ6’s output, still carrying the two dictionaries HJ5 emitted, is HJ7’s build side. So the dictionaries reach HJ7 by way of HJ6, and the surviving join activates there a second time, on the very same two dictionaries, narrowing the same two columns again over 109,842 rows. One emission at HJ4 supplies both narrowings, at HJ5 and at HJ7, with the perfect hash join between them preserving the dictionaries rather than breaking the chain.

What makes the result as large as it is comes down to what those two captured columns hold. One of them carries mini-biography prose, and after the query’s filter its values average 2,178 bytes each across the 12,595 distinct biographies. The build side materializes 1,618,081 rows, and on the native path every one of them copies its biography into the row store’s heap. That is 3.5 gigabytes of copies for twelve and a half thousand distinct values.

The optimization narrows the slot to a two-byte index into a dictionary that holds each biography once, and copies nothing. The operator falls from 249.26 to 18.54 milliseconds, a 13.4 times speedup.

The saving is almost entirely heap copies avoided rather than slot bytes narrowed, and this holds across the benchmark. Table 7.7 separates the two components.

query	join	layout (MB)	heap (MB)	heap share
7c	<code>linked_movie_id = movie_id</code>	45.3	3541.8	98.7%
7c	<code>id = movie_id</code>	3.1	269.7	98.9%
13b	<code>id = movie_id</code>	13.8	23.9	63.4%
13c	<code>id = movie_id</code>	13.8	23.9	63.4%
13d	<code>movie_id = movie_id</code>	6.6	10.7	61.8%

Table 7.7: The two components of the saving. *Layout* is the narrowed fixed-width slot, *heap* the non-inline string bytes the row store no longer copies, and *heap share* the heap’s fraction of the two.

On every one of these joins the avoided heap outweighs the narrowed layout, and by a wide margin. The reason is where a string lives. DuckDB stores a string of up to twelve bytes inline inside its fixed-width slot. A longer one does not fit, so the row store puts the bytes in a heap block beside the rows and leaves a pointer in the slot. Materializing such a value on the native path means following that pointer and copying its bytes into the hash table’s own heap, once for every build row the value appears on. Each row pays a variable-length copy rather than a fixed-width store, and the bytes moved grow with both the length of the string and the number of rows. On 7c’s first join the narrowed slot saves 45 megabytes, while not copying the biographies that slot pointed at saves 3542, almost 99 percent of the total. Across the suite the split is the same, 121 megabytes of layout against 3909 of heap, so 97 percent of what the optimization saves is heap copies it never makes. Narrowing the slot to a dictionary index removes them entirely, because the string already exists once in the dictionary and the row only needs to point at it.

The heap’s share follows the inline threshold directly. The 13 family narrows title columns short enough that many values fit inline, where they never touch the heap, so its heap share sits around 62 percent. Query 7c narrows biography prose averaging 2,178 bytes a value, all of it non-inline, so every value the native path handles is a heap copy and its heap share is 99. The longer the strings, the more of the saving is copies avoided rather than slot bytes narrowed.

7.2.6 Dictionary-aware probing

The probing optimization of Chapter 6 activates on 53 joins across 41 queries. On the 35 of those joins that take the dictionary path it eliminates 5,622,063 hash-table lookups, 52 percent of the 10,785,748 rows those joins probe. Table 7.8 reports the queries where it eliminates most.

query	joins	full probe	lookups	eliminated	% of full probe
31c	2	1,558,839	7,228	1,537,068	98.60%
25c	2	2,608,011	105,982	1,701,737	65.25%
18c	2	4,297,941	183,025	1,923,621	44.76%

Table 7.8: The three queries where the optimization eliminates most, each across the two joins where it activates. *Full probe* is every row those joins process, and the elimination is measured against it.

All three queries owe most of that elimination to one join they share, whose build side is six rows. The hash table holds only six distinct keys, so the cache that maps each key to

7. EVALUATION

its resolved bucket fills after the first few probe chunks and stays warm for the rest. That join sees 1,533,909 probe rows and issues 365 lookups, eliminating 99.98 percent of them, with 720 of its chunks warm. A warm chunk costs no hashes and no lookups at all.

The operator gain is real and it reproduces, close to 17 percent on the same join in all three queries. The queries themselves change little. Table 7.9 shows why.

query	join (ms)		query (ms)		reduction	
	disabled	enabled	disabled	enabled	join	query
31c	7.82	6.48	316.5	311.2	17.2%	1.7%
25c	7.80	6.47	388.4	386.8	17.1%	0.4%
18c	7.74	6.45	389.8	389.2	16.7%	0.2%

Table 7.9: The join the optimization speeds up, against the query that contains it.

The join it accelerates costs under eight milliseconds of a query that runs for over three hundred. Seventeen percent of two and a half percent is four tenths of a percent, and that is what the query gains. The joins this optimization can help are the small ones.

Dictionary-aware probing is limited by where the dictionaries sit. This optimization needs one on the probe key, while the surviving join needs one on a build-side payload column. On IMDB the dictionaries land on the payload columns far more often than on the keys, so the optimization applies to only a small share of the joins it could. A producer that emitted the key column as a dictionary as well, and not only the payload columns, would widen that share.

7.2.7 What the benchmark shows

The contributions are not independent. The emission produces 141 of the 193 dictionaries the surviving join captures, including every one behind the largest result. On its own it is a small operator-level win; as the supplier for the join that follows it, it is what makes that result possible. The two have to be measured together.

On real string data almost all of the saving is the heap copy the optimization no longer makes. Ninety-seven percent of what the surviving join saves is heap copies, and on the largest join the copies are 99 percent of the saving and the narrowed slot the remaining 1 percent. Measuring only the slot would therefore miss almost all of what this optimization does.

Dictionary-aware probing has the smallest effect of the three here. It cuts about 17 percent off the operator on its best join, but that join is a small part of its query, so the end-to-end gain is small. It activates only where a dictionary lands on the probe key rather

than a payload column, which on IMDB is rare. A dictionary from the storage layer serves it equally well, but DuckDB dictionary-encodes string columns on disk and not integer ones. Widening that encoding would widen where this optimization applies.

7.3 TPC-H

7.3.1 The benchmark

TPC-H is a different kind of test from the Join Order Benchmark. Its data is generated rather than observed, and it carries less skew than any other workload in this chapter, its columns holding either uniformly distributed values or a small set of distinct ones [47]. Its schema is eight tables in third normal form [39]. Its foreign key joins are on numeric attributes only. The benchmark is not short of string data, since strings are about 60 percent of the database by size [11]. What it lacks is string join keys, and that is what decides which of the three contributions it exercises. It exercises them narrowly. The benchmark was run at three scale factors, SF1, SF10 and SF100, spanning a hundredfold range of data, single-threaded and with the optimizer fully enabled.

7.3.2 Activation

Activation is sparse and shifts with scale. Table 7.10 reports which query activates each contribution, one table per scale factor.

The counts move between scale factors because the planner chooses different build sides as table sizes change, not because the data grows. The surviving join narrows a different set of columns at each scale, and the emission activates wherever a small build side meets a large probe. Dictionary-aware probing never runs at any scale, for a reason that is built into the schema and is taken up below.

7.3.3 Results

One query, Q21, gains end to end at every scale; the rest of the suite moves little. Those smaller gains are operator-local, visible at the hash join but diluted by the rest of the query. Peak memory is where the surviving join shows its clearest effect.

Table 7.11 shows this at each scale factor, for the queries that activate. Q21 is the one query that gains end to end at every scale, while peak memory moves only for the surviving join, and only where the narrowed column carries real bytes.

7. EVALUATION

SF1			
query	emission	surviving join	probing
Q5		•	
Q7		•	
Q9		•	
Q18		•	
Q19	•		
Q20	•		
Q21		•	

SF10			
query	emission	surviving join	probing
Q2	•		
Q5		•	
Q7	•		
Q8	•		
Q10	•		
Q17	•		
Q18	•		
Q19	•		
Q21		•	

SF100			
query	emission	surviving join	probing
Q2	•		
Q7	•		
Q8	•		
Q10	•		
Q15	•		
Q17	•		
Q18	•		
Q19	•		
Q21		•	

Table 7.10: Activation at each scale factor. Only the queries that activate at least one contribution are listed. Probing never activates at any scale. The emission activates on more queries as the data grows while the surviving join activates on fewer, both plan effects rather than scale effects.

7.3 TPC-H

SF1			
query	active	wall Δ	mem. Δ
Q5	surviving join	+0.44%	−0.3%
Q7	surviving join	−1.03%	−5.2%
Q9	surviving join	−0.10%	−2.6%
Q18	surviving join	0.0%	0.0%
Q19	emission	−3.79%	0.0%
Q20	emission	−0.17%	0.0%
Q21	surviving join	−1.54%	−13.1%

SF10			
query	active	wall Δ	mem. Δ
Q2	emission	−0.28%	0.0%
Q5	surviving join	−1.17%	−1.8%
Q7	emission	+0.24%	0.0%
Q8	emission	+0.97%	0.0%
Q10	emission	+0.20%	0.0%
Q17	emission	+0.68%	0.0%
Q18	emission	−0.30%	0.0%
Q19	emission	−1.17%	0.0%
Q21	surviving join	−1.36%	−8.9%

SF100			
query	active	wall Δ	mem. Δ
Q2	emission	−0.21%	0.0%
Q7	emission	−0.22%	0.0%
Q8	emission	−2.00%	0.0%
Q10	emission	+0.45%	0.0%
Q15	emission	−0.07%	0.0%
Q17	emission	+0.77%	0.0%
Q18	emission	−0.03%	0.0%
Q19	emission	+0.26%	0.0%
Q21	surviving join	−1.19%	−8.2%

Table 7.11: End-to-end wall time and peak memory change at each scale factor, for the queries that activate. A negative value is a saving. Q21 gains end to end at every scale; peak memory moves only for the surviving join.

One difference from IMDB is worth stating. On IMDB the emission and the surviving join met on the same query, the emission producing a dictionary that the surviving join then captured downstream. On TPC-H that never happens. No query activates both at any scale factor, so the two are never measured working together here, and each stands on its own.

7.3.4 The dictionary-surviving join

The surviving join is the one contribution that consistently pays on TPC-H. Query 21 is where it pays most, and it does so at every scale, with a small but steady end-to-end speedup of 1.54, 1.36 and 1.19 percent at SF1, SF10 and SF100. Q21 groups by `s_name`, a supplier name from the tiny `supplier` dimension, and threads it through a chain of large `lineitem` semi- and anti-joins. The surviving join narrows that column at three joins along

7. EVALUATION

the chain, and all three narrow the same dictionary.

That dictionary is not produced by the emission. It is created by a perfect hash join lower in the plan, the join on `s_suppkey`, whose key is a dense integer over a bounded range and so qualifies for a perfect hash table. Building that table, DuckDB emits each of its payload columns as a pipeline-global dictionary, including `s_name`. The three joins above it key on order and supplier keys that carry `s_name` only as a payload, which is exactly what the surviving join compresses. So one perfect hash join builds the dictionary once, and the surviving join captures it three times as it rides up the chain, re-emitted at each join rather than materialized back. This is the same survival mechanism as IMDB’s 7c, sourced from a perfect hash join rather than the emission.

The payoff is concentrated in the first capture. At SF100 that join takes 15,146,702 build rows, each carrying an `s_name` payload, and replaces every 16-byte string slot with a 4-byte index into the million-entry dictionary. The narrowed layout alone saves 181.8 megabytes, and the avoided heap copies of the non-inline strings save a further 272.6, so the operator falls from 4087.8 to 3851.5 milliseconds, a reduction of 5.78 percent and 236 milliseconds. This is the single largest saving on TPC-H. Across all three of Q21’s narrowed joins the total is 207.6 megabytes of layout and 311.4 of heap, 518.9 megabytes at SF100.

Q21 is instructive next to IMDB’s 7c on where that saving comes from. Its `s_name` values are eighteen bytes, just over the twelve-byte inline limit, so they are non-inline and the heap copy applies, but they are short. The heap saving is only about one and a half times the layout saving, where on 7c’s two-thousand-byte biographies it was seventy-eight times. Q21 is the case where layout and heap are comparable, 7c the case where the heap dwarfs everything. The longer the string, the more of the saving is heap.

The benefit does not scale with data in percentage terms, but its absolute value grows sharply. The first join saves about one millisecond at SF1, eleven at SF10, and 236 at SF100, while the percentage stays between four and six percent. The surviving join’s saving tracks the number of narrowed-slot gathers it performs, which grows with the rows flowing through the join. More data means more rows to narrow and a larger byte and millisecond saving, even as its share of a larger query holds roughly constant. Peak memory tells the same story, a steady reduction on Q21 of 13.05, 8.85 and 8.18 percent across the three scales, worth 674.5 megabytes at SF100.

7.3.5 The small build-side dictionary emission

On IMDB the emission was mainly a producer. It produced the dictionaries that the surviving join captured downstream, and most of its value came from that role rather than

from its own operator gains. On TPC-H that role disappears. No query activates the emission and the surviving join together, so nothing downstream ever captures an emitted dictionary. Every dictionary the emission produces is materialized back to values almost immediately, at the next operator that reads the column, and only the emission’s small operator-local effect remains.

That effect is decided by the width of the chunks the emission emits, because it replaces a value copy per row with a single pointer install per chunk, spreading its benefit over the rows in a chunk. Query 10’s emitting join produces full vectors, emitting 1,499,999 rows at SF10 and 14,999,999 at SF100. At both scales it is faster with the emission on, by 5.28 percent at SF10 and 4.43 percent at SF100. This is an operator-local gain that does not survive to the query. Without a downstream consumer the emission is a standalone optimization on TPC-H, and a modest one.

7.3.6 Dictionary-aware probing

This contribution never activates on TPC-H, at any scale factor. The reason is structural. The optimization needs a probe key that arrives as a dictionary vector, and TPC-H join keys are plain integers read flat from storage. The schema has no dictionary-encoded key column anywhere, so there is never an integer key it can turn into a compressed probe. DuckDB’s storage layer dictionary-encodes string columns but not integer ones, so an integer key read from storage always arrives flat rather than as a dictionary vector. If a future version dictionary-encodes integer columns too, TPC-H’s join keys could arrive compressed and the optimization could begin to activate. The dictionaries the workload does produce, the ones the emission and the perfect hash joins produce, sit on payload columns rather than keys, so none of them ever reaches a probe side where this optimization could use it. This is not a tuning problem. TPC-H simply does not contain the shape the optimization was built for, and a workload with dictionary-encoded join keys would be needed to exercise it.

7.3.7 What the benchmark shows

The surviving join is the one that pays, and it pays in memory more than in time. It improved its joins at every scale and produced TPC-H’s largest single saving on Q21 at SF100. Its value grows with data in absolute terms even as its share of the query stays small, and most of the saving is the heap copy avoided on non-inline strings rather than

7. EVALUATION

the narrowed slot. On a workload with longer strings the balance would tip further toward the heap.

The other two contributions are governed by the workload’s shape rather than its size. The emission is mainly a producer, most useful when a surviving join downstream captures the dictionary it produces. No TPC-H query activates the two together, so every emitted dictionary is materialized back with no consumer, and only a small operator-local gain remains. Dictionary-aware probing needs a dictionary-encoded probe key, which TPC-H’s plain integer keys never provide, so it never activates. Neither is a failure of the optimization. Each is a case of a real workload not containing the shape the optimization depends on.

7.4 TPC-DS

7.4.1 The benchmark

TPC-DS is larger and more complex than TPC-H. It is built from multiple snowflake schemas with shared dimensions, and holds 24 tables of 18 columns on average against TPC-H’s eight [39]. Its data is deliberately skewed, where TPC-H’s is close to uniform [39]. It shares the one property that decides this study. Every dimension’s primary key is a plain integer, and those integer keys are what the queries join on [55]. Its dimension tables are small by design, since fact tables scale linearly with the scale factor while dimensions scale sub-linearly [39]. The benchmark was run at two scale factors, SF1 and SF10, single-threaded with the optimizer fully enabled. Each query was run interleaved with the optimizations enabled and disabled, so drift cancels between the two runs, and Q51, which activates none of the three, was run alongside as a noise control.

7.4.2 Activation

Table 7.12 reports how many of the 103 queries activate each contribution at each scale.

SF1			SF10		
contribution	queries	joins or columns	contribution	queries	joins or columns
emission	23	27 joins	emission	29	40 joins
surviving join	14	77 columns	surviving join	13	59 columns
probing	1	1 join	probing	1	1 join

Table 7.12: Activation totals across the 103 queries at each scale factor. The emission and surviving join activate broadly; probing engages only one query at either scale, and it produces no result.

The emission and the surviving join activate on many queries, and their counts are stable across the tenfold change in data. Probing does work on one query at either scale, and even there it produces no result, for a reason taken up below. As on TPC-H, broad activation here does not turn into a broad result.

7.4.3 Results

One query clears the noise floor end to end, Q70, through the emission rather than the surviving join. Every other activated query stays within noise.

query	active	SF1	SF10
Q70	emission	4.39%	7.43%

Table 7.13: The only query whose end-to-end wall time clears the noise floor, at either scale.

The peak memory savings show a larger effect. Table 7.14 ranks the queries where it falls most at each scale.

SF1				
query	disabled	enabled	saved	reduction
Q40	29.3	25.1	4.2	14.3%
Q75	100.2	92.6	7.6	7.6%
Q64	109.8	102.0	7.8	7.1%
Q80	106.6	100.8	5.9	5.5%
Q78	245.7	237.1	8.7	3.5%
SF10				
query	disabled	enabled	saved	reduction
Q40	34.5	30.2	4.3	12.4%
Q84	36.4	33.7	2.6	7.2%
Q75	855.8	809.6	46.2	5.4%
Q78	2311.3	2251.8	59.5	2.6%
Q80	596.2	583.4	12.8	2.2%

Table 7.14: Peak memory in megabytes for the queries where the surviving join saves most, ranked by reduction. The suite total is 34 megabytes at SF1 and 138 at SF10, 0.6 and 0.4 percent of the respective footprint.

The absolute saving grows with data, from 34 to 138 megabytes across the suite. On individual queries the reduction reaches 14.3 percent, though it never converts into wall time. TPC-H showed the same operator-local pattern.

7. EVALUATION

7.4.4 The small build-side dictionary emission

The emission produces an end-to-end result on only one query, Q70. Its winning join is a store dimension joined to the store-sales fact table. The build side is the store dimension, which is tiny, 12 rows at SF1 and 102 at SF10. The probe side is the fact table, which is large, and nearly every fact row carries a store key that matches one of those few dimension rows. The join therefore emits about one output row per fact row, 536,455 of them at SF1 and 5.37 million at SF10.

The columns the emission works on are the store dimension's payload, the descriptive store attributes each fact row picks up through the join, such as the store's state and county. Without the emission each output row copies its own value of these attributes, so the same handful of store strings is copied again and again across millions of rows. The attributes take only a handful of distinct values across those millions of rows, exactly what a dictionary is for. The emission builds the dictionary once for the whole join and earns that cost back on every row that references it, so the 536,455 rows at SF1 and 5.37 million at SF10 all share a single build.

This is exactly the case the emission is built for, and it delivers. The join runs 18.9 percent faster at SF1 and 21.8 percent faster at SF10 with the emission on, and that one join is enough to speed up the whole query, by 4.39 percent at SF1 and 7.43 percent at SF10.

No other query gives the emission this shape. Its payoff is proportional to the rows a join emits, and most joins where the small build-side dictionary emission activates on TPC-DS emit few rows. Q72 is the clearest case against, where the emission builds a dictionary over a 62,322-row build side and the join then emits only 2,032 rows. The build cost is never spread, so it stays pure overhead. Only Q70 emits enough rows to repay the build, and only Q70 becomes a result.

7.4.5 The dictionary-surviving join

The surviving join activates on more columns than any other contribution here, 77 at SF1 and 59 at SF10, and it narrows real bytes on every one. It reduces peak memory on the queries it touches, by 14.3 percent on Q40 and by several percent on a handful of others, but none of them affect the wall time of the query. The reason is what it narrows. On TPC-DS the columns that ride up through the joins are plain integer keys, order numbers and ticket numbers and item keys, stored as eight-byte integers. Narrowing one to a two

or four-byte dictionary index saves a few bytes of row-store layout per row and touches no heap at all, because an integer key carries no heap.

This is the exact inverse of IMDB. There the surviving join narrowed long strings, and almost all of the saving was the heap copy avoided, up to seventy-eight times the layout on the largest join. Here the saving is almost entirely layout, 99 percent of it at both scales, because a fixed-width integer never touches the heap for the optimization to avoid. The two workloads sit at opposite ends of the same axis. On IMDB the saving is mostly the heap copy avoided, and it is large; on TPC-DS it is almost all the narrowed slot, and it is small.

The byte savings do scale. Summed across all the queries the surviving join activates on, the net saving, heap and layout together, grows from 14.2 megabytes at SF1 to 122.6 at SF10, close to linear on ten times the data. These byte savings are not large enough to reduce the end-to-end wall time of queries this size.

7.4.6 Dictionary-aware probing

Dictionary-aware probing does any real work on only one query, Q95, and it produces no end-to-end result there. Across the full join it eliminates 2.73 percent of the hash table lookups at SF1 and 1.51 percent at SF10, and Q95's wall time does not move.

The reason is the shape of the data. Probing needs its probe keys to arrive dictionary-encoded, and on TPC-DS they almost never do. Even on Q95, the one query where the optimization does anything, more than 96 percent of the chunks reaching that join are not dictionary vectors at all and are rejected before probing sees them. What it works on is the small remainder. Plain integer keys give it little to work with, and the queries that would exercise it are not in this suite.

7.4.7 What the benchmark shows

On TPC-DS almost all of the saving is the narrowed slot, with no heap-allocated data to avoid. The surviving join narrows integer keys rather than strings, so 99 percent of what it saves is fixed-width row-store layout and almost none is heap. This is the mirror image of IMDB, where the saving was 97 percent heap, and it is the clearest statement of what these contributions depend on. The size of the win is set by what the narrowed column holds, and an integer key holds little to save.

The one end-to-end result on TPC-DS comes from the emission, on the one query that emits enough rows to repay its dictionary build, and it stays a single query at both scales.

7. EVALUATION

Everywhere else the small build-side dictionary emission activates on joins that produce too few rows to pay. Probing needs its probe keys to arrive dictionary-encoded, and on this schema they almost never do, so it reaches only a small remainder of one query and produces no result. On a snowflake schema of integer keys the contributions activate widely and narrow real bytes, and almost none of it reaches the wall clock.

Conclusion and Future Work

8.1 Conclusion

An analytical database system holds a query’s working data in memory, which is expensive and is expected to take a growing share of what a server costs [57]. Queries fail or slow down when they run out of it, and the operators that materialize their input are one cause. A hash join copies its whole build side into a hash table, which is often the structure that sets a query’s peak memory. Building that table in compressed form is the goal of this thesis. Dictionary encoding can compress it, but only if one dictionary covers the whole column. A join pins a dictionary on its first batch and stores an index into it in every row. Storage builds one per row group and cannot promise that. A join can. It has its whole build side in hand before it emits a row, so the dictionary it emits describes every value the column takes. We call such a dictionary pipeline-global, and it is the main conceptual contribution of this thesis. Three optimizations follow from it, and together they let a column stay encoded across a chain of joins. We answer the research questions of Chapter 1 in turn.

RQ1. *How can a hash join produce dictionary-encoded output rather than materializing build-side values for every row?*

The small build-side dictionary emission adds one pass after the build side is finalized. That is the first moment at which the whole build side is known and its records sit together in the row store. The pass gathers each build-side output column into an array in row-store order. One position refers to the same record in every array. A single index per record therefore stands for all of its columns. That index is stored in the record’s next-pointer field, which after finalize is read only to walk the chains that

8. CONCLUSION AND FUTURE WORK

duplicate keys form. Where such chains exist, the original pointers are saved into an auxiliary array first, so no record grows and the row-store layout is unchanged. At probe time the join emits those columns as dictionary vectors sharing one selection vector. Each dictionary carries an identifier fixed for the life of the join, which is what makes it pipeline-global. The preparation is paid once, so the optimization runs only on a build payload under 8.4 megabytes. The probe must also hold at least 262,144 rows and be at least a hundred times the build. The saving then grows on two axes. It grows with the rows the join emits, since each one is a value copy avoided for every column projected. It grows with the payload too. More columns mean more copies on each row, and a wider column type means more bytes in each copy. The worst case is the preparation alone, a millisecond at most, recovered by the rows the join goes on to emit. On the standard workloads its main role is to feed the dictionary-surviving join with pipeline-global dictionaries.

RQ2. *How can a dictionary-encoded column be carried through the build side of a hash join in its compressed form?*

The dictionary-surviving join narrows a payload column's row-store slot to an index into that column's dictionary. It can do that only if the dictionary will not change. The sink fixes its slot widths on the first build chunk, and every later chunk has to resolve against the same dictionary. We call that guarantee pipeline-global and have the producer declare it, so the per-row-group dictionaries storage emits are never admitted by accident. Two operators give it, the perfect hash join and the emission above. The hash table used to fix its row-store layout in its constructor, from logical types alone. We therefore split its initialization in two and defer that decision. The first thread to see a build chunk chooses each column's slot width and publishes one layout for the others. Every thread then writes at the same byte offsets. A column narrows to a one, two or four-byte index according to the size of its dictionary. That happens only when the index is narrower than the column's native slot, so a row can never grow. The join pins the dictionary as a deep copy it owns. The producer's non-inline strings are bare pointers into a heap that is recycled before this join probes. Every build chunk then scatters indices rather than values. At probe the join reads each matched index back and re-emits a dictionary vector under the identifier it received. The dictionary therefore survives the join. A caching operator between the producer and the sink gathers small chunks into fuller ones. It rebuilt each column as

plain values, which broke the pipeline-global guarantee the producer had just given. We adapted it to maintain pipeline-global dictionaries through the buffer instead.

RQ3. *How can a hash join exploit a dictionary-encoded probe key to resolve each distinct value only once?*

Dictionary-aware probing resolves each dictionary slot once rather than each row. The join keeps a cache holding one hash-table chain head per slot, bound to the dictionary's identifier. On the first chunk under that identifier the cache is cold. The join collects the slots the chunk references and resolves that set against the hash table in one batch. It then copies each slot's head to every row that reads it. A later chunk resolves only the slots it is the first to see. Once every slot is resolved the cache is saturated, and the join stops looking for new ones. From there a chunk costs one pass that copies pointers, with no hashing and no lookups. The optimization activates only when the probe key arrives as a dictionary vector, and the ordinary probe runs otherwise. Correctness comes from machinery already in place. A lookup returns the head of a chain, so the per-row chain walk is untouched. Null keys pass through the filter the ordinary probe already uses. How long the cache lasts depends on where the dictionary came from. A storage dictionary rotates at every row-group boundary, while a pipeline-global one holds for the whole scan.

RQ4. *How much code change do these optimizations require in DuckDB, and is that footprint small enough for the change to be maintainable in a production engine?*

The three optimizations add 1,238 lines and remove 105, excluding tests and benchmarks. The small build-side dictionary emission accounts for 239 added and 52 removed across three engine files. The dictionary-surviving join is the largest at 693 and 53 across ten. Dictionary-aware probing adds 306 and removes none across three. All three were reviewed and merged into DuckDB. We therefore consider them maintainable changes for a production engine.

RQ5. *What effect do these optimizations have on analytical query execution?*

RQ5.1. *What is their effect on query performance?*

The gain is concentrated in a few queries. On the Join Order Benchmark query 7c ran 2.23 times faster, and queries 13b, 13c and 13d gained between 6 and 11 percent. The suite as a whole moved 1.4 percent. TPC-H gained on Q21 at every scale factor, by less than two percent. TPC-DS gained on one query, by

8. CONCLUSION AND FUTURE WORK

4.39 and 7.43 percent at the two scales. The joins themselves gained more. The emission cut operator time by up to 20.5 percent, and dictionary-aware probing by 17 percent on its best join. A join is one part of a query, so most of that does not reach the wall clock.

RQ5.2. *What is their effect on memory use?*

Memory is where the dictionary-surviving join pays. On the Join Order Benchmark query 7c fell by 91.1 percent, from 4311 to 385 megabytes. Queries 13b, 13c and 13d fell by between 27 and 33 percent. Across the suite peak memory fell 13.4 percent, almost all of it that one query. On TPC-H query 21 fell by 13.05, 8.85 and 8.18 percent at the three scales, worth 674.5 megabytes at the largest. On TPC-DS the reduction reached 14.3 percent on one query. The reduction depends on how many rows the hash table holds and on the width of the payload columns narrowed. We observed the best results on long strings, which do not fit inline and are copied to the heap for every build row.

RQ5.3. *What is their effect on the operators downstream of the join?*

A dictionary emitted by one join survives the build of the next and leaves it under the same identifier, so one dictionary can outlive several operators. On the Join Order Benchmark query 7c carries one dictionary from the join that emitted it into two later joins. A perfect hash join between them passes it on untouched. On TPC-H query 21 carries a dictionary built by a perfect hash join through three joins as it rides up the plan. Where the next operator is not dictionary-aware, the column is materialized back to values at once and the compression buys nothing beyond the join. Where it is, the saving continues past the join. On a query whose group-by sits above a surviving join, a projection and a hash aggregate that were not modified for this work saved between them nearly as much as the join itself.

8.2 Future work

8.2.1 More dictionary-aware operators

A dictionary reaching an operator is not the same as an operator using it, as the experiments of Chapter 4 show. Several operators could do better, and none of them needs the dictionary to be pipeline-global. The ungrouped aggregate is the first of them, and could

work per distinct value rather than per row. DuckDB's grouped aggregate already resolves a dictionary once per distinct value, but only when the query groups on a single column.

8.2.2 More operators using the pipeline-global guarantee

The pipeline-global guarantee means one dictionary covering the whole of the column its producer emits, unchanged from one chunk to the next. Work done once against it therefore never has to be redone. A sort, a window operator and a filter are three that could benefit from that. None reads a dictionary today, and each repeats work per row that depends only on the distinct values.

8.2.3 Compressing the key as well as the payload

The dictionary-surviving join narrows payload columns and leaves the key at its native width. The difference is that a payload slot is written and later read, while a key slot is written and later compared. A key could be stored as an index too, with the comparison resolving the real value out of the dictionary. That trades processor work for memory. The row store shrinks, while each key comparison gains a lookup. The join could then also emit the key column as a dictionary vector rather than materializing it.

8.2.4 The dictionary-surviving join with storage dictionaries

Storage is the common source of dictionaries, and the dictionary-surviving join cannot take one. A scan builds a new dictionary as it advances, and a code from one means nothing in the next. The sink would have to hold every dictionary it is handed, either pinning each one separately or unifying them into a space of its own. Each of those has its own problems to solve.

8.2.5 A second dictionary guarantee

The pipeline-global dictionary is a property a producer declares rather than a mechanism it implements. Distinctness is a second property of the same kind, that no two entries hold the same value. A consumer could then compare codes where it now compares values. An equality filter could find its constant in the dictionary once, and a join could compare stored keys without resolving them. Today the two properties do not meet. Storage dictionaries are distinct and change as the scan advances, while the pipeline-global ones never change and can hold a value twice. Deduplicating the emission would give DuckDB its first dictionary that is both.

8. CONCLUSION AND FUTURE WORK

References

- [1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. “The Design and Implementation of Modern Column-Oriented Database Systems”. In: *Foundations and Trends in Databases* 5.3 (2013), pp. 197–280. DOI: 10.1561/19000000024 (1, 5, 7–9, 11, 12, 22, 26, 40, 58).
- [2] Daniel J. Abadi. “Query Execution in Column-Oriented Database Systems”. Ph.D. dissertation. Massachusetts Institute of Technology, Feb. 2008 (18).
- [3] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira. “Integrating Compression and Execution in Column-Oriented Database Systems”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006, pp. 671–682. DOI: 10.1145/1142473.1142548 (2, 11, 12, 22, 23).
- [4] Omid Afroozeh. “Unified String Dictionary in DuckDB”. Joint UvA-VU Master of Science degree in Computer Science. MSc thesis. Vrije Universiteit Amsterdam, Aug. 2025 (28, 29, 59).
- [5] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. “Amazon Redshift Re-invented”. In: *Proceedings of the 2022 International Conference on Management of Data, SIGMOD 2022, Philadelphia, PA, USA, June 12-17, 2022*. ACM, 2022, pp. 2205–2217. DOI: 10.1145/3514221.3526045 (8, 23).
- [6] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. “Photon: A Fast Query Engine for Lakehouse Systems”. In: *Proceedings of the 2022 International Conference*

REFERENCES

- on Management of Data, SIGMOD 2022, Philadelphia, PA, USA, June 12-17, 2022*. ACM, 2022, pp. 2326–2339. DOI: 10.1145/3514221.3526054 (8).
- [7] Peter Boncz, Stefan Manegold, and Martin Kersten. “Database Architecture Optimized for the New Bottleneck: Memory Access”. In: *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999*. Morgan Kaufmann Publishers Inc., 1999, pp. 54–65 (29).
- [8] Peter Boncz, Thomas Neumann, and Viktor Leis. “FSST: Fast Random Access String Compression”. In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2649–2661. DOI: 10.14778/3407790.3407851 (2, 8, 12, 28, 31).
- [9] Peter Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution”. In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005*. www.cidrdb.org, 2005, pp. 225–237. URL: <https://www.cidrdb.org/cidr2005/papers/P19.pdf> (8, 9).
- [10] *The Technology behind CedarDB*. CedarDB documentation. CedarDB. URL: <https://cedardb.com/docs/technology/> (visited on 07/28/2026) (8).
- [11] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. “Query Optimization in Compressed Database Systems”. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA, May 21-24, 2001*. ACM, 2001, pp. 271–282. DOI: 10.1145/375663.375692 (22, 97).
- [12] *LowCardinality(T)*. ClickHouse documentation. ClickHouse. URL: <https://clickhouse.com/docs/sql-reference/data-types/lowcardinality> (visited on 07/28/2026) (12).
- [13] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. DOI: 10.1145/362384.362685 (5, 14).
- [14] Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. “SchemaPile: A Large Collection of Relational Database Schemas”. In: *Proceedings of the ACM on Management of Data* 2.3 (June 2024). DOI: 10.1145/3654975 (2).
- [15] *Quack Remote Protocol*. DuckDB project site. DuckDB. URL: <https://duckdb.org/quack/> (visited on 07/28/2026) (6).
- [16] *Storage Versions and Format*. DuckDB documentation. DuckDB. URL: <https://duckdb.org/docs/current/internals/storage> (visited on 07/28/2026) (7, 59).
- [17] *storage_info.hpp*. DuckDB source, commit be691253. DuckDB. URL: https://github.com/duckdb/duckdb/blob/be6912536d6230dbf1d949ce953f88721afdeea/src/include/duckdb/storage/storage_info.hpp (visited on 07/29/2026) (7, 13).

REFERENCES

- [18] *Execution Format*. DuckDB documentation. DuckDB. URL: <https://duckdb.org/docs/current/internals/vector> (visited on 07/28/2026) (9, 10, 12, 13).
- [19] *Overview of DuckDB Internals*. DuckDB documentation. DuckDB. URL: <https://duckdb.org/docs/current/internals/overview.html> (visited on 07/28/2026) (9).
- [20] *dictionary_vector.hpp*. DuckDB source, commit f7c47484. DuckDB. URL: https://github.com/duckdb/duckdb/blob/f7c4748410fc54b2ff51a01ba31bba4e06f827d9/src/include/duckdb/common/vector/dictionary_vector.hpp (visited on 07/29/2026) (13).
- [21] *Vectors*. DuckDB C API documentation. DuckDB. URL: <https://duckdb.org/docs/current/clients/c/vector.html> (visited on 07/28/2026) (15).
- [22] *aggregate_hashtable.cpp*. DuckDB source, commit dd54fb63. DuckDB. URL: https://github.com/duckdb/duckdb/blob/dd54fb63a27cf839cb9fd2135d324ef62f6a196d/src/execution/aggregate_hashtable.cpp (visited on 07/29/2026) (18, 30, 66).
- [23] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. “The SAP HANA Database – An Architecture Overview”. In: *IEEE Data Engineering Bulletin* 35.1 (2012), pp. 28–33 (24, 25, 30).
- [24] Yannis Foufoulas, Lefteris Sidiourgos, Eleftherios Stamatogiannakis, and Yannis Ioannidis. “Adaptive Compression for Fast Scans on String Columns”. In: *Proceedings of the 2021 International Conference on Management of Data, SIGMOD ’21*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 554–562. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3452798 (24, 26, 28).
- [25] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. “SQLite: Past, Present, and Future”. In: *Proceedings of the VLDB Endowment* 15.12 (2022), pp. 3535–3547. DOI: 10.14778/3554821.3554842 (6).
- [26] Bogdan Ghiță, Diego Tomé, and Peter Boncz. “White-box Compression: Learning and Exploiting Compact Table Representations”. In: *10th Annual Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, Netherlands, January 12–15, 2020*. www.cidrdb.org, 2020. URL: <https://vldb.org/cidrdb/2020/white-box-compression-learning-and-exploiting-compact-table-representations.html> (23).

REFERENCES

- [27] Neha Gholkar, Jovan Stojkovic, Hasan Al Maruf, Gregory Price, Prakash Chauhan, Hiral Patel, Cedric Van Goethem, Kiran Vemuri, Kiran Malwankar, Kishore Sriadibhatla, Kalyan Subramanian, Shobhit Kanaujia, Chunqiang Tang, and Abhishek Dhanotia. “Vistara: Making CXL Real – Full Path from ASIC Design and OS Support to Hyperscale Deployment”. In: *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2026, pp. 703–717. DOI: 10.1109/ISCA66397.2026.00061 (1).
- [28] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135. DOI: 10.1109/69.273032 (7).
- [29] Goetz Graefe and Leonard D. Shapiro. “Data Compression and Database Performance”. In: *ACM/IEEE-CS Symposium on Applied Computing, Kansas City, MO, April 1991*. IEEE Computer Society Press, 1991, pp. 22–27. DOI: 10.1109/SOAC.1991.143840 (1, 2, 11, 12, 18, 21, 31).
- [30] Paul Groß, Daniel ten Wolde, and Peter Boncz. “Adaptive Factorization Using Linear-Chained Hash Tables”. In: *15th Annual Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025. URL: <https://vldb.org/cidrdb/2025/adaptive-factorization-using-linear-chained-hash-tables.html> (15–17).
- [31] Tim Gubner, Viktor Leis, and Peter Boncz. “Optimistically Compressed Hash Tables & Strings in the USSR”. In: *SIGMOD Record* 50.1 (Mar. 2021), pp. 60–67. DOI: 10.1145/3471485.3471500 (2, 25, 28, 30, 32, 33, 57–59).
- [32] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. “How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans”. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 2007, pp. 389–400. DOI: 10.1145/1247480.1247525 (30).
- [33] Laurens Kuiper, Paul Groß, Peter Boncz, and Hannes Mühleisen. “Saving Private Hash Join”. In: *Proceedings of the VLDB Endowment* 18.8 (2025), pp. 2748–2760. DOI: 10.14778/3742728.3742762 (14–17, 34).
- [34] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, San Francisco, California, USA, June 26 - July 1, 2016*. ACM, 2016, pp. 311–326. DOI: 10.1145/2882903.2882925 (22, 23, 26, 27).

REFERENCES

- [35] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. “Real-Time Analytical Processing with SQL Server”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1740–1751. DOI: 10.14778/2824032.2824071 (23, 25, 30).
- [36] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. “Joins on Encoded and Partitioned Data”. In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1355–1366. DOI: 10.14778/2733004.2733008 (2, 14, 26, 27, 31–33, 39).
- [37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. “How Good Are Query Optimizers, Really?” In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215. DOI: 10.14778/2850583.2850594 (3, 32, 88).
- [38] Alicja Łuszczak. “Simple Solutions for Compressed Execution in Vectorized Database System”. Joint master thesis with Vrije Universiteit Amsterdam, Faculty of Sciences. Master’s thesis. University of Warsaw, Faculty of Mathematics, Computer Science and Mechanics, July 2011 (1, 2, 6, 11–13, 24, 25, 30, 31, 40, 58, 59).
- [39] Raghunath Othayoth Nambiar and Meikel Poess. “The Making of TPC-DS”. In: *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB ’06, Seoul, Korea, September 12-15, 2006*. VLDB Endowment, 2006, pp. 1049–1058 (97, 102).
- [40] Thomas Neumann and Michael Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *10th Annual Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, Netherlands, January 12-15, 2020*. www.cidrdb.org, 2020. URL: <https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html> (8, 15, 16).
- [41] Mosha Pasumansky and Benjamin Wagner. “Assembling a Query Engine From Spare Parts”. In: *CDMS 2022, 1st International Workshop on Composable Data Management Systems, Sydney, Australia, September 9, 2022*. 2022 (8).
- [42] Mark Raasveldt. *Move to Push-Based Execution Model*. DuckDB issue #1583. DuckDB, 2021. URL: <https://github.com/duckdb/duckdb/issues/1583> (visited on 07/29/2026) (9).
- [43] Mark Raasveldt. *DuckDB: In-Process Analytical Database System*. Guest lecture slides, 15-721 Advanced Database Systems. Carnegie Mellon University, 2023. URL:

REFERENCES

- <https://15721.courses.cs.cmu.edu/spring2023/slides/22-duckdb.pdf> (visited on 08/03/2026) (10, 11).
- [44] Mark Raasveldt. *Push-Based Execution in DuckDB*. Seminar talk. Dutch Seminar on Data Systems Design. URL: https://dsdsd.da.cwi.nl/past_talks/duckdb-push-based-execution/ (visited on 07/29/2026) (9).
- [45] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212 (3, 6–9, 14).
- [46] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1080–1091. DOI: 10.14778/2536222.2536233 (24, 25, 31, 33, 34).
- [47] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. “Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet”. In: *Proceedings of the VLDB Endowment* 17.11 (2024), pp. 3694–3706. DOI: 10.14778/3681954.3682031 (1, 2, 88, 97).
- [48] Mark A. Roth and Scott J. Van Horn. “Database Compression”. In: *SIGMOD Record* 22.3 (Sept. 1993), pp. 31–39. DOI: 10.1145/163090.163096 (12).
- [49] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. “ClickHouse - Lightning Fast Analytics for Everyone”. In: *Proceedings of the VLDB Endowment* 17.12 (2024), pp. 3731–3744. DOI: 10.14778/3685800.3685802 (3, 8).
- [50] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. “Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, Arizona, USA, May 20-24, 2012*. ACM, 2012, pp. 731–742. DOI: 10.1145/2213836.2213946 (24–26).
- [51] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. “C-Store: A Column-oriented DBMS”. In: *Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005*. 2005, pp. 553–564 (22).

REFERENCES

- [52] TrendForce. *Rapid Contract Price Surge Drives 1Q26 DRAM Industry Up 81% QoQ*. Press release. 2026. URL: <https://www.trendforce.com/presscenter/news/20260601-13070.html> (visited on 08/08/2026) (57).
- [53] Eames Trinh. “Improving Parquet Compression Using Global Dictionaries in Delta Lake”. Joint UvA-VU Master of Science degree in Computer Science; work carried out at Databricks. MSc thesis. Vrije Universiteit Amsterdam, July 2025 (25, 26).
- [54] *Dictionary Encoding*. Velox documentation. Velox. URL: <https://facebookincubator.github.io/velox/develop/dictionary-encoding.html> (visited on 07/28/2026) (3, 10, 12).
- [55] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. ACM, 2018, 1:1–1:6. DOI: 10.1145/3209950.3209952 (2, 88, 102).
- [56] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. “Building An Elastic Query Engine on Disaggregated Storage”. In: *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 449–462. URL: <https://www.usenix.org/conference/nsdi20/presentation/vuppalapati> (1).
- [57] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. “TMO: Transparent Memory Offloading in Datacenters”. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022*. ACM, 2022, pp. 609–621. DOI: 10.1145/3503222.3507731 (57, 107).
- [58] Yifan Wu, Yuhan Li, Zhenhua Wang, Zhongle Xie, Dingyu Yang, Ke Chen, Lidan Shou, Bo Tang, Liang Lin, Huan Li, and Gang Chen. “SafeLoad: Efficient Admission Control Framework for Identifying Memory-Overloading Queries in Cloud Data Warehouses”. In: *Proceedings of the VLDB Endowment* 19.4 (2025), pp. 713–725. DOI: 10.14778/3785297.3785311 (1).
- [59] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. “An Empirical Evaluation of Columnar Storage Formats”. In: *Proceedings of the VLDB Endowment* 17.2 (2023), pp. 148–161. DOI: 10.14778/3626292.3626298 (1, 28).

REFERENCES

- [60] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. “SAHA: A String Adaptive Hash Table for Analytical Databases”. In: *Applied Sciences* 10.6 (2020), p. 1915. DOI: 10.3390/app10061915 (34, 35).
- [61] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. “Super-Scalar RAM-CPU Cache Compression”. In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*. IEEE, 2006. DOI: 10.1109/ICDE.2006.150 (25, 26, 28, 29, 59).
- [62] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. “Vectorwise: A Vectorized Analytical DBMS”. In: *2012 IEEE 28th International Conference on Data Engineering, ICDE 2012*. IEEE, 2012, pp. 1349–1350. DOI: 10.1109/ICDE.2012.148 (8).