University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

VU University Amsterdam
Faculty of Sciences

Kamil Anikiej

Student no. 235894(UW), 2002612(VU)

Multi-core parallelization of vectorized query execution

Master thesis
in COMPUTER SCIENCE

First reader
Henri Bal
Dept. of Computer Science,
Vrije University

Second reader
Peter Boncz
Centrum Wiskunde & Informatica
Vrije University

Supervisor
Marcin Żukowski
VectorWise B.V.

July 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data   Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data   Podpis autora (autorów) pracy

Abstract

Due to the recent trends in computer hardware, especially processors, parallel query execution is becoming more and more important in modern database management systems (DBMS). In this master thesis, we discuss the implementation of a parallel query execution system based on a new family of operators that was firstly presented in the Volcano DBMS. We combine this approach with a highly efficient vectorized in-cache execution model used in the VectorWise DBMS. We present different strategies of incorporating new operators into an execution tree. Finally, we propose possible optimizations and measure the performance of implemented solutions.

Keywords

parallel query execution, multi-core, in-cache execution, cost model, Volcano model

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

D.127.6. Concurrent Programming: Parallel Programming
H.2.4 Systems: Query Processing

Contents

Chapter 1

Introduction

1.1. Motivation

Constant development of computer hardware requires new approaches to software design. New performance opportunities and performance bottlenecks have to be identified in all layers of the hardware stack (super-scalar CPUs, cache memories, main memory, disk). This also applies to query processing in relational database systems.

In this thesis, we focus specifically on the VectorWise database management system. VectorWise is a novel DBMS designed to fully exploit modern hardware architectures. Its main areas of applicability are query-intensive applications such as on-line analytical processing or data-mining.

The VectorWise DBMS is already capable of inter-query parallelism i.e. it can work on several queries concurrently. However, parallelization of a single query (intra-query parallelism) is not currently supported. This is a significant drawback, which, when considering the common availability of multi-core CPUs, limits performance capabilities of this DBMS. This leads to the objective of this thesis:

Objective. *Design and implement efficient intra-query parallelism in the VectorWise DBMS.*

In this master thesis we

(i) describe an implementation of the Xchange family of operators that introduce intra-operator parallel query execution in the VectorWise DBMS,

(ii) develop strategies and transformation rules to rewrite a given non-parallel query execution tree into its optimal or close to optimal parallel counterpart,

(iii) measure performance of the implemented solutions.

We present a novel approach to incorporating the vectorized execution model (see section 2.2.1) into the so-called *Volcano model* (sec. 2.3.1). We also show the applicability of our design in the VectorWise DBMS by measuring its performance in the TPC-H benchmark [TPC].

Chapter 2

Query processing

2.1. DBMS architecture

A simplified organization of a relational DBMS consists of:

client application - which issues the SQL query,

query parser - which builds an internal representation of the parsed query (a tree),

query normalization - which checks for the semantical correctness of the tree,

query rewriter - which rewrites the query tree into another which is estimated to have the lowest execution cost,

query executor - in which we construct all the operators (the *building* phase) and actually processes the data,

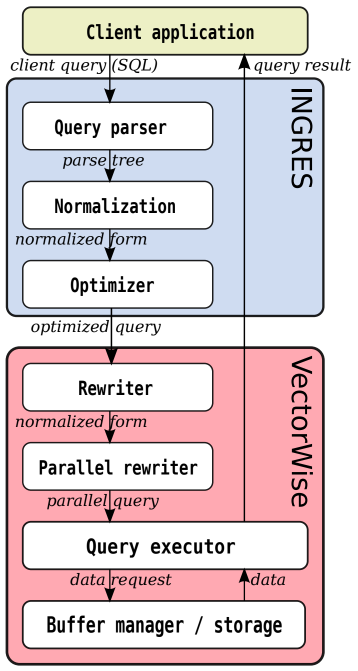
buffer manager / storage - which handles storing data on persistent media.

The organization of the VectorWise DBMS is a little bit more complicated. This DBMS consists mainly of the execution engine (the *query executor* layer). It does not parse user's SQL queries, but uses an open-source DBMS (Ingres) for that purpose (figure 2.1.1a).

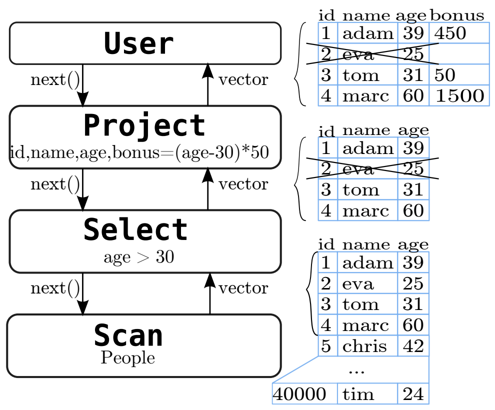
After parsing the SQL query, and checking for its correctness, Ingres DBMS generates an optimized *query execution tree* or *QET* (see the next section) and passes it to the VectorWise DBMS. In order to do that, Ingres keeps track of all the tables and indices created by a user. It also maintains statistics (e.g. cardinality, value distribution) about the stored data.

The *rewriter* module in the VectorWise DBMS is responsible for preparing and optimizing query execution trees for processing. For example, it assigns the data types to the relational attributes, introduces lazy evaluation of some expressions or eliminates dead code.

From the perspective of this master thesis, two components are of the most importance. The *parallel rewriter* and the *query executor*. In the parallel rewriter (from now on abbreviated to *rewriter*) we construct a parallel query execution plan, which is afterward processed by the query executor module.



(a) A simplified architecture of the VectorWise DBMS. Ingres DBMS is used to transform SQL queries into a query execution tree.



(b) A simple query execution tree with the Project, Select, Scan operators. Operators return vectors (of the size of 4) of data as a result of the *next()* call.

Figure 2.1.1: Query processing in the VectorWise DBMS.

2.2. VectorWise DBMS

The VectorWise DBMS stresses data analysis (data warehousing, reporting, information retrieval etc.) as its main area of applicability. It is designed to exploit the performance potential of modern computers. Contrary to typical database engines it benefits from new processor features such as SSE, out-of-order execution, chip multi-threading or increasingly larger L2/L3 caches. The VectorWise DBMS uses a vectorized in-cache execution model.

2.2.1. Vectorized execution model

Most database engines use the iterator model in their query execution layers with a standard *open()*, *next()*, *close()* interface. Queries are represented as a tree of operators, where the operators are taken from the Relational Algebra, e.g. Scan, Select or Join. During the evaluation of a query tree tuples are pulled up through the query tree, by calling *next()* on the query root (which leads to subsequent *next()* calls to on its children, etc.). A query plan thus consists of a set of relational operators that operate in a “pipeline” fashion (figure 2.1.1b).

Traditional database management systems operate on a single tuple at a time. Because the *next()* method is called to produce a single tuple, which usually represents just a few bytes of data, instructions related to query interpretation and tuple manipulation outnumber data manipulation instructions. It increases the instructions-per-tuple ratio. Moreover, for every tuple there are multiple function calls performed, which results in a low instruction-per-cycle factor. The reason for this low efficiency is that query plans are processed at run-time. Hence, function calls needed for interpretation are late-binding function calls, as query plans in all their details are (C/C++) objects that are linked together at run-time. Late-binding function calls, in turn, impede many of the most profitable compiler optimizations such as inlining function bodies, loop optimizations, etc. Absence of such optimizations cause a code pattern where modern CPUs achieve only low instruction per cycle throughput

To avoid the overhead present in the tuple-at-a-time model, the VectorWise DBMS uses a vectorized in-cache execution model, which reduces the instructions-per-tuple cost. In this model, operators work on *vectors* of values (one dimensional arrays of values, e.g. 1024 values), which are the basic data manipulation and transfer units.

Primitives

In the VectorWise DBMS all operations on data are performed using short and highly specialized functions called *primitives*. This DBMS defines a large number of those components [*]. Each primitive implements code that is easy to optimize for compilers and which fully exploits modern CPUs features like SIMD instructions, pipelined execution, branch prediction, or hardware prefetching.

Below we provide an example of a primitive routine that adds two vectors of integers and checks for the overflow [Żu09].

```
void map_add_int_vec_int_vec(int *result, uint *input1, uint *input2, int n) {
  ulong overflow = 0;
  for (int i = 0; i < n; ++i) {
    ulong l1 = input1[i];
    ulong l2 = input2[i];
    ulong res = l1 + l2;
    overflow |= res;
    result[i] = (int) res;
  }
  if (overflow > 0xFFFFFFFFUL) {
    return STATUS_OVERFLOW;
  }
  return STATUS_OK;
}
```

The *for* loop does not suffer from branch mispredictions and does not contain any data dependencies. It is easy to unroll and is an easy subject to SIMDization. Figure 2.2.1 presents the efficiency of such a primitive, which also partially answers the question about the optimal vector size. The best results are obtained for the sizes such that all vectors used in the query plan fit into the L1 cache memory [†].

[*]Thousands of such functions (for each operation and input types) are generated using macro expansions.
[†]The most frequently used vector sizes are 1024, 2048 or 4096 values.

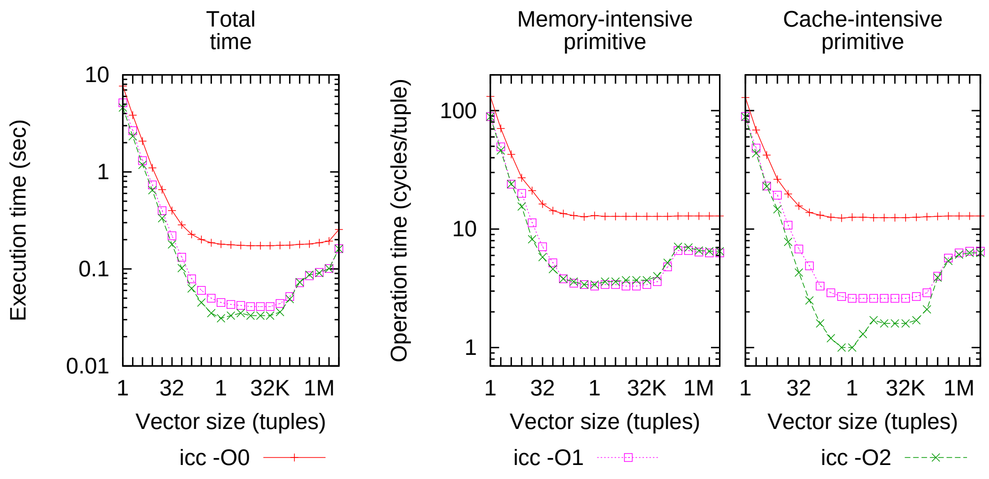


Figure 2.2.1: Primitive routine efficiency. Impact of the vector size and different optimizations on the time and the number of cycles per tuple. Courtesy of Marcin Żukowski.

2.2.2. Column Store

VectorWise is a column-oriented DBMS. The values of one column are stored consecutively on a disk (contrary to a row-oriented DBMS in which values of one record are consecutive). This design enables us to read only relevant data (only the necessary subset of all columns) and avoid interpretation overhead, which in row-oriented DBMS systems is mainly spent on extracting the data from a row.

Multiple disc accesses when adding or modifying a record are the major drawback of this representation [‡] . Therefore, column-oriented DBMS are mainly used for read-intensive large data repositories.

2.2.3. Cluster Trees

The VectorWise DBMS exploits the idea of *cluster trees*, which we introduce in this section. Cluster trees are not a tree-like data structure, rather refer to a data ordering strategy, in which we co-order tables that have foreign key relationship. The foreign key relationship paths, which can be clustered, form a tree that is a subset of the schema graph. Cluster trees allow us to build much more efficient query execution trees that replace some expensive hash join operations with their faster merge-based counterparts. In this section, we also present the concepts of join indices and join-range indices.

Join Index

The idea of join indices is introduced in [Val87]. Let A and B be two relations. A join index is an abstraction of the join of two relations. We assume that each tuple of a relation is uniquely identified by a so-called Stable ID (SID), We denote by a_i (b_i respectively) a tuple

[‡]This problem is partially solved by in-memory differential structures [HZN$^+$10].

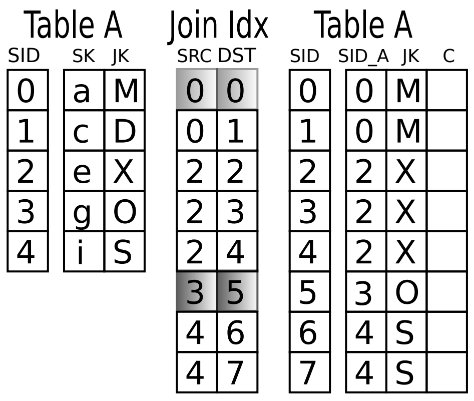


Figure 2.2.2: A traditional join index between Table A and Table B, with B being sorted on A through a foreign key relationship. Shaded values (samples of the whole join index) are the basis for a join-range index.

from relation A (resp. B), for which its SID equals i. Then, the join index on A and B is the set

$$JI = \{(a_i, b_i) \mid f(tuple\ a_i, tuble\ b_i) \text{ is true}\}$$

where f is a Boolean function that defines the join predicate.

Clustering Join Indices

While generally useful, join indices also have some limitations. Traditional DBMS exploit foreign key joins between tables having a join index using a nested loop index join. This method requires $log(N)$ random disk accesses for each tuple, which is unacceptable in the VectorWise DBMS. Other approaches i.e. using the merge join or the hash join algorithms also involve some drawbacks. In the former case, both input streams to the merge join have to be sorted, which is almost always slower than the hash join algorithm [§]. In the latter case, the problem of the hash table not fitting RAM arises. Table clustering solves these problems.

The idea of cluster trees is to store a table in the physical order corresponding to the tuple order in some other table. More formally, let A be a table sorted on some attributes (denoted as Sort Key or SK). Let B also be a table referencing A by means of a foreign key constraint. We say that B is *clustered* on A if B is kept in the sort order of A. An example of this situation together with a join index is depicted in Figure 2.2.2.

We can consider a table a node in a graph and the relation of being clustered (or having a join index between tables) as an edge. In this representation tables form a tree-like structure. This tree is called *cluster tree.*

Sometimes we do not need all the information stored in a join index and its approximation is also sufficient. If tables are clustered, then values in both columns of a join index are known to be non-decreasing. We can divide a join index into chunks and store only the information about minimum values (for SRC and DST columns) inside each chunk. We call

[§]This will probably change in the future. See section 3.5 and [KKL+09].

this structure a *join-range index*. The average size of a chunk can be a configurable parameter allowing us to store this structure in RAM. Since we have some freedom in specifying a set of chunks for a join-range index, we put one additional requirement on this structure:

Property. *Two consecutive chunks never separate tuples having the same value on JK attributes.*

This means that a chunk border never separates a group of the same values. This property is used later on in section 5.3.

2.3. Related work

Over the recent two decades significant efforts have been made in order to design efficient algorithms for parallelization. We can distinguish two major trends *(i)* algorithms based on sequential implementations and *(ii)* algorithms based on data partitioning.

Among the algorithms belonging to the first group, developed solutions reduce the number of both data and instruction cache misses, exploit capabilities of modern CPUs or eliminate the problem of data skew. For example [KKL$^+$09] focuses on two popular join algorithms – hash join and sort-merge join and presents an implementation that takes advantages of latest processor features.

An important contribution to the approach based on data partitioning, on which we base our solution, is the Volcano model proposed in early 90s [GCD$^+$91]. The idea of incorporating this concept into the VectorWise DBMS was firstly presented in [Żu09].

A vector-based partitioning was used in [HNZB07] to successfully parallelize a query without join operators. However, it is unclear whether this approach is efficient and general enough for queries with join operations.

[WFA95] discusses and evaluates four different strategies of parallel query execution of multi-join queries. The conclusions presented cannot be directly applied for our purposes as the tested approaches uses different (non-Volcano) execution models.

Cieslewicz *et al.* in [CRG07] proposes using *parallel buffers* for reducing both data and instruction cache misses. The Volcano model uses a similar approach, but also exploits the advantages of a pipelined execution analyzed in [BZN05].

A metric which balances the estimated query execution time and the total execution cost of a parallel query execution tree was proposed in [GHK92] and [vB89]. The cost calculus formulas were adapted and applied to our execution model from [GHK92].

A study on different thread synchronization libraries and their applicability for database systems under various usage scenarios is presented in [JPA08].

2.3.1. Volcano model

The proposed solution to parallelization of a single query is based on the Volcano model. In Volcano's design, all parallelism issues like partitioning and flow control are encapsulated in and provided by a new family of operators and are independent of data manipulation issues.

The module responsible for parallel query execution is called *Xchange*. It implements *(i)* a number of new operators (described in chapter 4) which conform to the same *open()*, *next()*, *close()* iterator interface and can be inserted at any place in a query tree and *(ii)* a procedure for finding the optimal parallel execution tree (chapters 5 and 6). Other operators are implemented and executed without regard to parallelism. This is an important advantage from the engineering point of view, as parallelism may be implemented orthogonally to other parts of the systems.

Chapter 3

VectorWise Algebra

In this chapter, we give a short introduction to a subset of relational query processing operators implemented in the VectorWise DBMS. A more detailed description of operators, their algorithms, optimization techniques (especially for large data sets) can be found in [Gra93]. A VectorWise specific description can be found in [Żu09].

In the following sections we use the term *column* as a synonym of attribute. We also introduce a concept of *streams*. Streams are anonymous inputs (an operator does not know what other operator produced its input or whether it was produced from a complex subtree or a simple table scan) that allow combining any number of operators to evaluate a complex query.

Operators form a query execution trees and operate in a demand-driven schema. The return value of the *next* call is a pointer to the vector of data and the size of the vector.

3.1. Project

The Project operator is used for *(i)* introducing new columns with values based on other columns, *(ii)* renaming columns, *(iii)* filtering entire columns. It exports only those attributes that were explicitly used in the projection list.

We use the following notation for the Project operator: $\text{Project}_{projlist}$ (if not required we omit the *projlist* parameter). The projection list is a comma-separated list of projections, where `<projection> ::= <column_id> | <column_id> '=' <expression>`.

Output: Columns explicitly named in the projection list.

3.2. Select

Selection is used to choose only those tuples that match a specified predicate (and filter out those that do not).

The Select operator is denoted by: $\text{Select}_{bool\_expr}$. The boolean expression *bool_expr* can be an arbitrary (possibly nested) expression returning a boolean result.

Output: Select propagates exactly the same columns as its child.

3.3. Aggregation

Aggregation provides an important statistical concept to summarize information about large amounts of data. The idea is to represent a set of items by a single value or to classify items into groups and determine one value per group.

The Aggregation operator is denoted by: $\text{Aggr}_{grpby\_list,aggregate\_list}$. The *grpby_list* is a list of attributes. If empty, global aggregation is performed. The *aggregate_list* is a list of named aggregate expressions, including *sum*, *min*, *max* and *count*. Average (*avg*) is not supported and has to be simulated by using the results of the *sum* and *count* functions in a post-computation (e.g. in the Project operator which is a parent of the Aggregation operator) The bodies of aggregate functions need to be simple columns, without any arithmetic. If *aggregate_list* is empty, a duplicate elimination is performed.

Output: Aggregation returns all the group-by keys and aggregate functions.

Commutative property of SQL's aggregating functions

If an Aggregation operator processes only a subset of data (this subset may be a result of data partitioning performed during parallelization) the output data carries only partial results. For example the *minimum* function is calculated for separate parts of the data and does not yield the global minimum. Those partial answers are obviously incorrect from the global point of view.

This problem can be solved if a common property of *max*, *min* and *sum* functions is exploited. Those functions are *commutative* so computing the global answer from partial results suffices.

For the *count* function we have to calculate the *sum* of its partial answers as the cardinality of disjoint subsets of data that sum up to the original set is the sum of partial cardinalities.

The only exception, *average* function, is tackled by postponing its computation and operating only on its necessary components - *sum* and *count* functions, which are commutative.

3.4. Ordered Aggregation

The OrderedAggregation operator is an optimized version of the Aggregation operator for the situation when the input stream is *clustered*. We say that a sequence $a_0, a_1, a_2, \ldots$ of data (e.g. a stream) is clustered if

$$\forall_{i \leq j} a_i = a_j \rightarrow \forall_{k \geq i \wedge k \leq j} a_k = a_i$$

i.e. when two tuples with the same values on the key columns are not separated by a tuple having a different value.

The OrderedAggregation operator is denoted by: $\text{OrdAggr}_{grpby\_list,aggregate\_list}$

Output: See the Aggregation operator.

3.5. HashJoin

The algorithm of the HashJoin operator builds a hash table with the values from the right (or inner) data stream. Then, the values from the left (or outer) stream are passed through the operator in a pipelined fashion. For each of the tuples a decision about whether to pass it further to the parent operator is based on the content of the hash table (the tuple may or may not have a matching counterpart in the build input). Concluding, the outer data stream is executed only after the hash table is fully computed (is materialized).

We say that a flow is *materialized* if it is fully processed before any tuple can be used by the parent operator. Materializing the data does not scale well. The intermediate data has to fit into RAM as otherwise a few orders of magnitude slower disk-resident swap memory has to be accessed.

Another processing schema, in which a tuple, after being processed by a given operator, can be instantly passed to the parent operator, is called a *pipelined* execution (see [BZN05] for a detailed analysis).

The HashJoin operator is denoted by: $\text{HashJoin}_{keys_A,keys_B}$. Two sets of columns (of the same cardinality) $keys_A$ and $keys_B$ define the relational operation.

Output: The hash operators generate tuples matching the relational operation. It outputs all attributes of the left operator plus all attributes of the right operator.

3.6. MergeJoin

The second commonly used join method is the merge-join. All merge-based joins assume two inputs ordered on the same sequence of keys, and perform operations on them based on the values of the key columns. Merging the two inputs is similar to the merge process used in sorting.

The MergeJoin operator is denoted by: $\text{MergeJoin}_{keys_A,keys_B}$.

Output: See the HashJoin operator.

3.7. Sort

Sorting is frequently used in database systems. In the VectorWise DBMS Sort operator is used mainly for presenting to the user sorted reports or listings and less often for query processing in sort-based algorithms such as merge-join. This can, however, change because of the increasing importance of sort-merge join operator, which better exploits modern CPU features and therefore may overtake HashJoin algorithms in the near future [KKL+09].

The Sort operator is denoted by: $\text{Sort}_{sort\_columns}$.

Output: Sort returns the same shape of the table as its input, with tuples sorted on the *sort_columns*.

3.8. TopN

TopN has the same behavior as the Sort operator. It is used to return the first N tuples from a flow according to some sorting order. It is denoted by: $\text{TopN}_{sort\_columns,N}$.

3.9. Reuse

The Reuse operator supports non-tree query graphs (DAGs). Reuse executes the underlying query only once, and buffers the tuples so they can be read by many consumers. This optimizes execution of query plans which have to recompute costly subtrees.

Chapter 4

Xchange operators

In this chapter we present the family of Xchange operators, which are solely responsible for introducing parallelism into a query execution tree. Each of those operators provides the same interface as any other operator i.e. a standard set of *open(), next(), close()* methods. Hence, from the implementation point of view, those operators are indistinguishable from any other operator. We can insert any of the Xchange operators into a query execution tree without having to change its parent or its child.

An Xchange operator provides a control flow in a parallel QET. It can split one data stream into many, each handled by a different thread, or join multiple streams into one.

From now on by a *stream* we understand a flow of data processed by a separate thread.

4.1. Examples of Intra-query parallelism

Before defining all the Xchange operators in more detail in this section, we present an example of intra-query parallelism with a XchgUnion operator. We focus on a simple query with an execution tree that consists of a single path only.

4.1.1. Parallelizing Aggregation

An SQL query that can be rewritten into the execution plan in figure 4.1.1a is presented below:

```
SELECT   returnflag,sum(extprice * 1.19) as sum_vat_price
FROM     lineitem
WHERE    shipdata <= '2000-01-01'
GROUP BY returnflag
```

In the execution plan, the Scan operator reads *returnflag*, *shipdata* and *extprice* columns from the *lineitem* table. The Select operator passes only those tuples that meet the "WHERE" condition. The Project operator calculates the new *sum_vat_price* column, whereas the Aggregation operator groups tuples that have the same value of *returnflag* and calculates the sum of their *sum_vat_price* values.

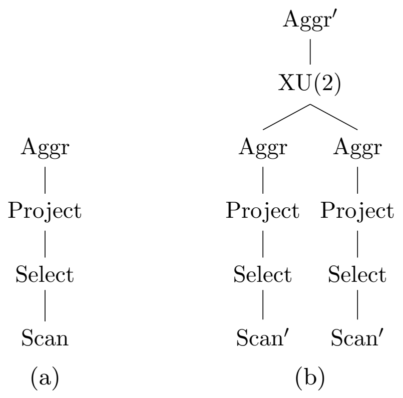


Figure 4.1.1: Parallelization introduces 2 additional operators, executes the Aggr-Project-Select-Scan path in parallel and reduces by half the scan range in the Scan operators.

For this specific query we know that the output (the number of returned tuples) of the Aggregation is small. This is because the *returnflag* column has a very limited set of distinct values, which can be checked in relation attribute's histograms[*]. Moreover, the histograms can indicate that the input to the Aggregation operator is large (i.e. there are many tuples meeting the `shipdata <= '2000-01-01'` condition).

In the described circumstances, the rewriter can decide to insert the XchgUnion operator on top of the Aggregation operator and to divide its execution subtree between its children. The XchgUnion operator consumes data from its subtrees and produces a single flow of data. The children operators are executed by different threads and operate on distinct subsets of data.

Depending on whether the input data is divided between the threads, we may or may not have to add additional Aggr operator on top of the XchgUnion operator. In the simplest approach the Scan operator divides its input ranges equally between threads as presented in (4.1.1).

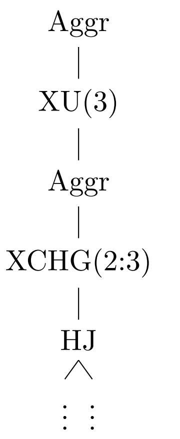
$$\text{range } [a..b) = \begin{cases} \text{range } [a \,..\, a + \frac{(b-a)*1}{n}) & \text{for the first thread} \\ \dots & \dots \\ \text{range } [\frac{a+(b-a)*(n-1)}{n}..b) & \text{for the } n\text{-th thread} \end{cases} \tag{4.1.1}$$

In another solution, the input is dynamically split and distributed on demand, which introduces additional overhead, but tackles the problem of load imbalances. A more sophisticated approach divides the input with respect to its content [†].

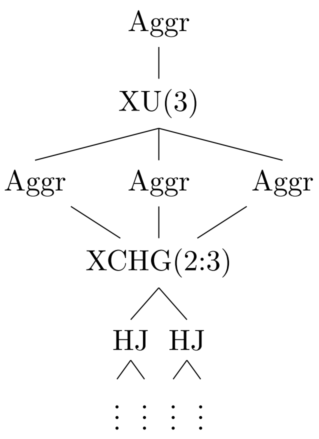
The lack of additional Aggregation operator (operator *Aggr'* in figure 4.1.1b) results in an incorrect plan. This error can be much easier noticeable for the `SELECT count(*) FROM lineitem` SQL query. Each Aggregation operator returns the cardinality of the data it

[*]Histograms are used to predict cardinality and the number of rows returned by a subtree. They are indispensable part of every data optimization process.

[†]The latter solutions are described in section 5.3.



(a) QET representation used. It is more concise, but implicit.



(b) Alternative notation with operators and streams shown explicitly. In this representation it is clear to see that each Aggregation and HashJoin operator are separate objects.

Figure 4.2.1: Different representations of operator tree with XchgUnion and Xchg operators.

operates on (a half of the whole input). As a result, the XchgUnion operator produces two answers (with values that sum up to the correct answer), instead of one (as expected). The new Aggregation operator computes the global answer out of two partial answers calculated only on a subsets of data. For details please refer to section 3.3.

4.2. Xchange operator definitions

In this section we define the operators from the Xchange family of operators.

4.2.1. Xchg(N:M) operator

The Xchg(N:M) operator provides a flow control in a QET. It transforms N data streams into M data streams, hence it is a synchronization point between $N + M$ threads.

Figure 4.2.1 presents two different ways of representing Xchg(N:M) operator in a QET.

In this document, we use the more concise 4.2.1a representation. Figure 4.2.1b presents how the same QET maps into an operator tree. From this perspective each stream is processed by an independent set of operators. The only exception are Xchange operators that work on and synchronize multiple streams.

In the example query, when traversing the tree in the top-down manner, we encounter the XchgUnion (or Xchg(3:1) - see the next section) operator that splits the processing between three Aggregation operators. Below, the Xchg operator changes the number of streams in the data flow from three to two, so that the workload is spread between two HashJoin operators.

4.2.2. XchgUnion (Xchg(N:1)) operator

Each query execution tree has to produce a single stream as its output. Therefore, this operator is present in every parallel tree as the top-most among Xchange operators. There is no difference between the XchgUnion and the Xchg(N:1) operators, but because of its importance it is given a special name.

4.2.3. XchgBroadcast (Xchg(1:M)) operator

This operator distributes the child's output to many consumers. All consumers are guaranteed to consume all the input data (contrary to any other Xchange operator) and to consume it in the same order as the data was produced. The XchgBroadcast operator is especially useful to assure that a given subtree in an execution tree is executed only once.

4.2.4. XchgHashSplit operator

Data produced by the XchgHashSplit operator is mutually disjoint between streams. This operator splits the data produced by its children in a content-aware manner. The decision which consumer a given tuple will reach is made with respect to the list of columns K passed as a parameter to this operator. As a result, we obtain a useful property of the output data:

Property. *If any tuples* $t1$ *and* $t2$ *from the operator's input are equal on columns from* K *they will reach the same consumer.*

As additional computation is performed, this operator has larger computation cost than its non-hashing counterparts.

4.2.5. XchgDynamicSplit

The XchgDynamicSplit operator is similar to the XchgBroadcast operator. However, each tuple is consumed only once. This operator is used to provide a load balancing mechanism for scenarios where children process data at different speed.

4.2.6. XchgDynamicHashSplit

The XchgDynamicHashSplit operator joins the idea of the XchgHashSplit (partitioning the input) and the XchgDynamicSplit (distributing a piece of data to only one consumer) operators.

4.2.7. XchgFork

XchgFork is an extended version of the Xchg(1:1) operator. XchgFork operators are paired, which provides new possibilities for the flow control. A pair of XchgFork operators is used together with binary operators, where it is put between the operator and its children. The

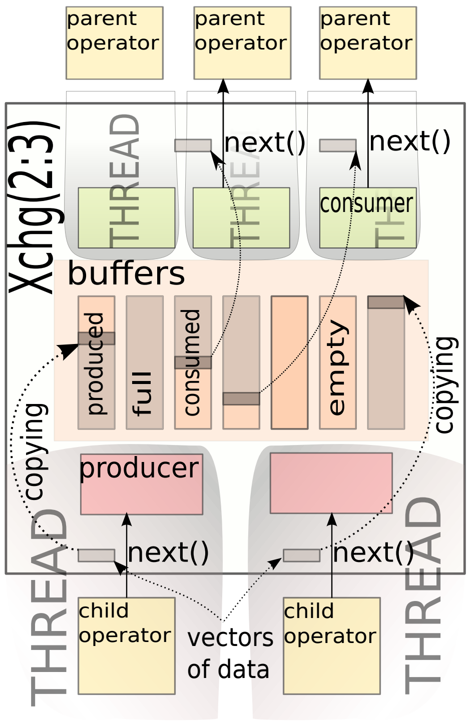


Figure 4.3.1: Xchg(2:3) operator.

first *next*() call issued on any of the XchgFork operators from a given pair, triggers execution of *both* subtrees of a binary operator.

4.3. Xchange operator - implementation

4.3.1. Producer-Consumer schema

All Xchange operators comprise of producer(s) and consumer(s), which are separate logical objects. An Xchange operator is responsible for creating and managing a number of buffers (typically proportional to the number of consumers times the number of producers), which temporarily store the data consumed from the children operators. A new thread is created for each of the producers. This implies that also every consumer operates in a separate thread (because it is either the main thread or it is a (possibly non-direct) child of some producer).

Let us now focus on figure 4.3.1 which presents the internal structure of the Xchg(2:3) operator. This schema can be analyzed together with the example presented in figure 4.2.1b (the *parent* operators become the Aggregation operators and the *children* operators become the HashJoin operators).

This Xchange operator is a synchronization point between five threads (two producers and three consumers). Child operators return vectors of data (as the output of the *next*() function calls), which are processed by producers and copied into local buffers. Producers are responsible for choosing the destination for the new vector of data and for the copying process. Once a vector is processed, a producer may issue a new *next*() call on its child. In the meantime, consumers (when asked for in the *next*() call issued by the parent operator) return the data stored in buffers. This is an $O(1)$ operation consisting only of setting the pointers to the data.

4.3.2. Producer - description

Algorithm 1 Producer part 1 - acquiring buffers
Once started, each producer acquires first buffer(s). The synchronization routines (mutex_lock(), mutex_unlock(), cond_wait(), cond_signal()) used in the listings, are the standard *Pthread* implementation, although [JPA08] proposes different libraries, which may result in better performance.

```
1:  $num\_buffers \leftarrow$ num_produced_buffers($producer$)
2:  mutex_lock($lock$);
3:  for $i \leftarrow 0$ TO $num\_buffers$ do
4:      $buffer[i] \leftarrow$ producer_find_empty_buffer($producer$);
5:      while $buffer[i] == nil$ do
6:          cond_wait($producer\_cond$, $lock$)
7:          $buffer[i] \leftarrow$ producer_find_empty_buffer($producer$);
8:      $buffer[i].state \leftarrow PRODUCED$
9:      $buffer[i].write\_pos \leftarrow 0$
10:     $buffer[i].for\_consumer \leftarrow i$
11: mutex_unlock($lock$);
```

Firstly, each producer has to obtain its initial buffers (listing 1). In the first line, a producer determines the number of buffers it will maintain at once (those buffers will be marked as PRODUCED). For hashing operators (e.g. XchgHashSplit) the $num\_buffers$ variable becomes the number of consumers as a given tuple my reach any of the consumers. Producers in other Xchange operators maintain only one buffer at a time.

Afterward, each producer performs a loop with the following operations (listing 2):

(i) calls *next*() method on its child operator (line 3), the *data* variable becomes the pointer to the data returned by the child operator, whereas the n variable becomes the size of the data returned (it may be smaller than the default vector size as some operators (e.g. Select, Aggregation) do not always produce a full vector). If n equals zero the data flow has finished (line 5),

(ii) in line 6 hashing operators determine the addressee of each tuple form the *data* vector. This information is stored in *selection_vector*. Consumer i will receive the following tuples:

$$\{selection\_vector[i][0], \ldots, selection\_vector[i][vector\_count[i]-1]\}$$

(iii) if any of the producer's buffers is full, a consumer is notified and producer tries to obtain a new empty buffer (lines 9 to 31),

(iv) the data is copied into buffers that are now known to have enough free space (line 33).

Algorithm 2 Producer part 2 - the main producer's loop
A generic producer's code shared by all operators from the Xchange family of operators. Specific implementations differ only in the *release_buffer* method.

```
Require: Producer owns $buffers[0]$ to $buffers[num\_buffers - 1]$
    which are obtained using Algorithm 1.
1: loop // The main producers loop
2:   $num\_buffers\_relased \leftarrow 0$
3:   $n, data \leftarrow child$.next();
4:   if $n == 0$ then
5:     break from inner loop
6:   $vector\_count[], selection\_vector[] \leftarrow$process($data$, $n$)
7:
8:   for $i \leftarrow 0$ TO $num\_buffers$ do
9:     if $buffer[i].write\_pos + vector\_count[i] > BUFFER\_CAPACITY$ then
10:      if $num\_buffers\_released == 0$ then
11:        mutex_lock($lock$)
12:      $num\_buffers\_released \leftarrow num\_buffers\_released + 1$
13:      release_buffer($buffer[i]$, $producer$);
14:      $buffer[i] \leftarrow nil$;
15:  if $num\_buffers\_released > 0$ then
16:    cond_broadcast($consumer\_cond$)
17:    loop // Loop until all buffers are acquired
18:      $num\_buffers\_to\_wait \leftarrow 0$
19:      for $i \leftarrow 0$ TO $num\_buffers$ do
20:        if $buffer[i] == nil$ then
21:          $buffer[i]$ = producer_find_buffer($producer$);
22:          if $buffer[i] == nil$ then
23:            $buffer[i].write\_pos \leftarrow 0$
24:            $buffer[i].state \leftarrow PRODUCED$
25:          else
26:            $num\_buffers\_to\_wait \leftarrow num\_buffers\_to\_wait + 1$
27:      if $num\_buffers\_to\_wait == 0$ then
28:        break from inner loop
29:      else
30:        cond_wait($producer\_cond$, $lock$)
31:    mutex_unlock($lock$)
32:  for $i \leftarrow 0$ TO $num\_buffers$ do
33:    copy_data_into_buffers($buffer[i]$, $selection\_vector[i]$, $vector\_count[i]$)
34: mutex_lock($lock$);
35: $num\_working\_producers \leftarrow num\_working\_producers - 1$
36: for $i \leftarrow 0$ TO $num\_buffers$ do
37:   release_buffer($buffer[i]$, $producer$);
38: cond_broadcast($consumer\_cond$);
39: mutex_unlock($lock$);
```

4.3.3. Consumer - description

A consumer in the *next*() method performs (Algorithm 3):

(i) if consumer has not acquired any buffer yet or consumer's old buffer has been consumed, it tries to obtain a full buffer (line 1)

   (a) if a buffer is consumed by all its addressees it is marked as EMPTY and a producer is notified (lines 6 to 9),

   (b) consumer searches for a new buffer (line 13)

   (c) if a there is no buffer ready for consumption but at least one producer is working consumer retries the attempt after some time (lines 14 to 16)

   (d) consumer obtained a new buffer and marks it as consumed (lines 21 to 24)

(ii) the buffer is being consumed - the data is returned from the currently owned buffer (lines 28 to 31).

4.3.4. Buffers - description

An ANSII C definition of a buffer is given below.

```
typedef struct {
  Expr **data;            /* Data placeholder. */
  BufferState state;      /* EMPTY, FULL, PRODUCED, CONSUMED. */
  slng product_id;        /* Time of production. */
  slng write_pos;         /* Producers writing pointer. */
  slng for_consumer;      /* Addressee. */
  slng consumers_done;    /* # of consumers that finished
                             reading from this buffer. */
} Buffer;
```

Each buffer is owned either by a producer or by a consumer (or possibly many consumers). If a buffer is owned by a producer, the access is exclusive. However, a single buffer may be concurrently read by many consumers (this happens for example in the XchgBroadcast operator).

A buffer may be in four different states:

- *EMPTY* - the buffer is not owned by any consumer nor producer. It also does not hold any data. This is the initial state of every buffer. An empty buffer may be obtained by a producer.
- *PRODUCED* - the buffer is being written into. It is exclusively owned by a producer, which copies its child's data into the buffers empty area (indicated by *write_pos*).
- *FULL* - the buffer was *PRODUCED*, but the producer is unable to find enough free space to write a new vector of data. The buffer is marked as *FULL* and waits for a consumer.
- *CONSUMED* - the buffer is being read from. It is owned by at least one consumer, which passes data to its parent (which is an $O(1)$ operation). Then each consumer updates its current *read_pos* pointer.

Algorithm 3 Consumer - implementation of the operator's *next*() method.
A generic consumer's code shared by all operators from the Xchange family of operators. Specific implementations differ in *buffer_is_done*, and *consumer_find_buffer* methods.

```
if $buffer == nil$ OR $consumer.buffer\_read\_pos == buffer.write\_pos$ then
    // We have to acquire a new buffer
    mutex_lock($lock$);
    if $buffer \neq nil$ AND $consumer.buffer\_read\_pos == buffer.write\_pos$ then
        $buffer.consumers\_done \leftarrow buffer.consumers\_done + 1$
        if buffer_is_done($buffer$) then
            $buffer.state \leftarrow EMPTY$
            $buffer.consumers\_done \leftarrow 0$
            cond_signal($producer$)
    $buffer \leftarrow nil$

    if $buffer == nil$ then
        $buffer$ = consumer_find_buffer($consumer$)
        while $buffer == nil$ AND $num\_working\_produces > 0$ do
            cond_wait($consumer\_cond, lock$)
            $buffer$ = consumer_find_buffer($consumer$)
        if $buffer == nil$ AND $num\_working\_producers == 0$ then
            // All producers have finished. No more buffers will be created.
            mutex_unlock($lock$);
            return 0
        // Now $buffer \neq nil$
        $buffer.state \leftarrow CONSUMED$
        $consumer.next\_buffer \leftarrow buffer.product\_id + 1$
        $consumer.buffer\_read\_pos \leftarrow 0$
    mutex_unlock($lock$);

// Return data from $buffer$
$returned\_tuples \leftarrow buffer.write\_pos - consumer.buffer\_read\_pos$
$returned\_tuples \leftarrow MIN(returned\_tuples, vectorsize)$
set_pointers_for_returned_data($buffer$, $consumer.buffer\_read\_pos$)
$consumer.buffer\_read\_pos \leftarrow consumer.buffer\_read\_pos + returned\_tuples$
return $returned\_tuples$
```

Microbenchmarks

The optimal size and the number of the buffers is an interesting optimization problem. On the one hand, larger buffers reduce the number of synchronizations both for producers and consumers. On the other hand, large buffers materialize more data (deteriorating advantages of the pipelined execution) and consume more memory.

Also, a higher number of buffers balances temporal differences between the speed of producers and consumers (if at some point consumers are faster than producers they may obtain previously filled in buffers, also temporarily faster producers may benefit from a spare set of empty buffers). This reduces the chance of a producer or a consumer having to wait for a buffer in a specific state, but increases the memory usage.

Figure 4.3.2 presents the analysis of the optimal buffer size for a particular query, which consists of an Array-XchgUnion-NullOp query tree. Both the Array and the NullOp operators are implemented for debugging purposes only. The Array operator generates a sequence of values, whereas the NullOp operator consumes data, without processing it or passing it further. In this scenario, the majority of computation is performed by the XchgUnion operator, which is used to parallelize the Array operators. This is an extreme situation in which data production and parent operators are instantaneous. Nearly 100% CPU time is spent in the Xchange operator. In real queries the parent and child subtrees perform work. This means that the relative overhead of the Xchange operators is smaller.

The number of producers used times the amount of data a single thread processed was constant. The number of buffers was three times the number of producers. The horizontal axis presents the size of one buffer. The size is defined in vectors of 1024 values (e.g. 4096 values fit into the buffersize of 4). The test was performed on a machine with two quad-core 2.8GHz Nehalem processors; hence with 8 cores.

We also measured the impact of the size of the buffer in the *ideal* situation, which almost completely eliminates the overhead (which is already very small) of the consumer's parent operator. For this purpose we changed the implementation of the XchgUnion operator so the consumer instantly returns its buffer to the pool of empty buffers (returning only a fraction of the data from this buffer). This change results in an incorrect implementation, but allows as to measure the impact of a producer not having to wait for an empty buffer.

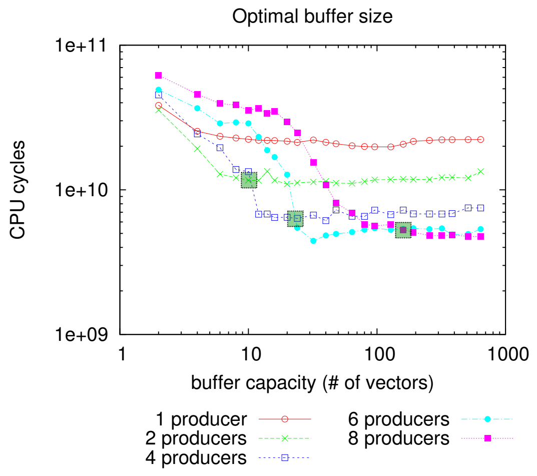
The execution time (measured in the number of CPU cycles) for this query is presented in figure 4.3.2a. Adding new threads resulted in better speedups, but was only possible if buffers were big enough. Increasing the level of parallelism results in additional overhead (thread locking, thread notifying, searching for a new buffer), which has to be compensated by less frequent synchronization (which is on a per-buffer basis). The green squares presents the minimum size of a buffer required by 4 (resp. 6, 8) producers to overtake 2 (resp. 4, 6) producers.

Figure 4.3.2b presents the number of thread preemptions caused by a failure in acquiring a lock. The lock contention decreases with larger buffers (as expected). There are also only small differences between 4, 6 and 8 threads. The explanation is that smaller execution time compensates for a higher probability of a lock being already acquired by a different thread.

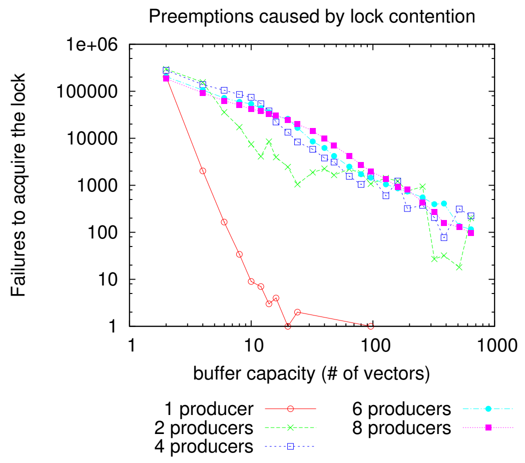
Figure 4.3.2c shows the average waiting time, which at some point increased by a factor of 5 after changing from 6 to 8 producers. In this benchmark, the consumer shared the CPU with a producer (there were 8 CPUs and 9 threads - 8 threads for producers, one thread for consumer). As a result, 8 producers were faster than a single consumer and had to wait for an empty buffer.

This load imbalance is clearly visible when comparing figures 4.3.2a, 4.3.2c and 4.3.2d. The consumer's parent operator (NullOp) is an operator performing hardly any computation. Nevertheless, even this simple operator introduces some overhead and makes the consumer slower. In the "ideal" situation we obtain better speedups for much smaller buffer sizes. Consumer is able to return empty buffers to the pool fast enough and producers do not have to wait for a buffer. Also, the average time required to acquire the lock is not as much dependent on the number of threads.

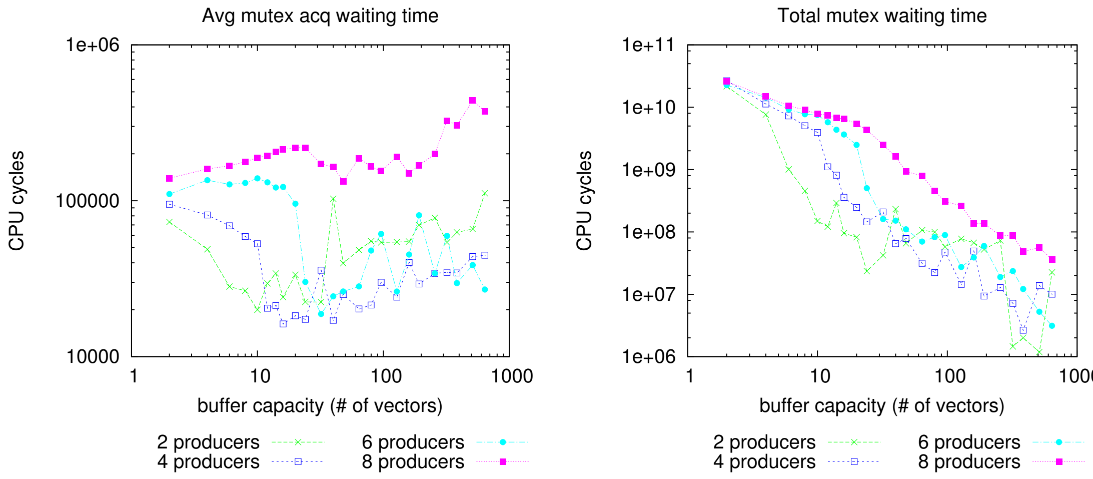
The most important conclusion derived from this benchmark is that the buffer size has to increase with the number of threads used. The execution time functions obtained their minimums in the proximity of the buffer size of 200 vectors (figure 4.3.2a the rightmost green square). Currently however, we support buffers of a fixed size only. In the implementation of the Xchange operators, because of the memory usage concerns we used buffers of the size of 50 vectors. Smaller buffers also decrease the start-up delay in the plans and does not impact the benefits of the pipelined execution [BZN05].



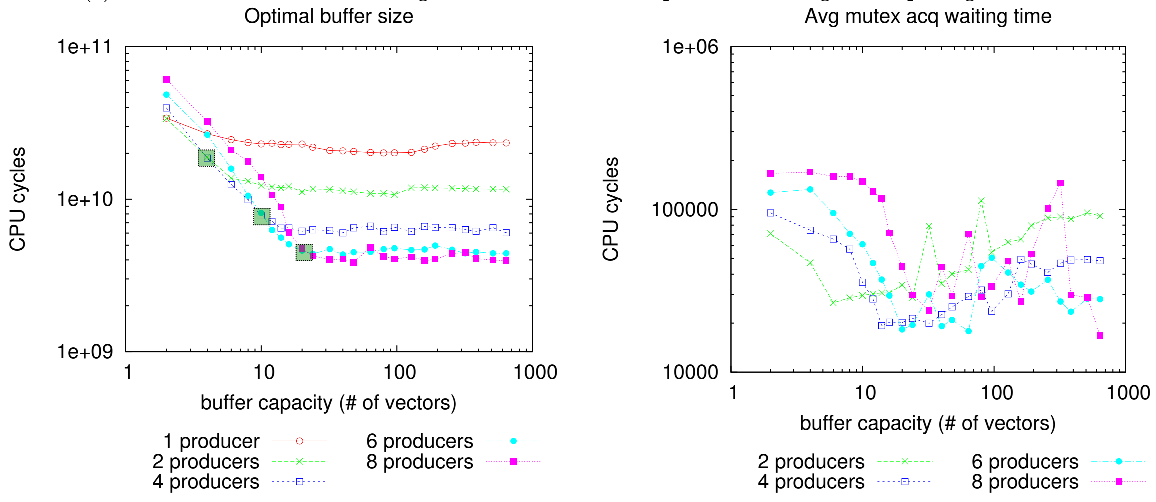
(a) Query execution time as a function of the buffer size. The green square indicates the minimum buffer size required by X threads ($X \in 4, 6, 8$) to overtake $X - 2$ threads.



(b) The number of failures in acquiring the lock (thread is preempted).



(c) Lock contention. The average and the total time spent on waiting for acquiring the lock.



(d) An experiment with a very fast consumer.

Figure 4.3.2: Impact of different buffer sizes on the performance (note: log-log scale used).

Chapter 5

Rewriter

5.1. Parallelizing Query Execution Trees

The number of all possible subtrees and all possible parallelizations for these trees is too large for exhaustive search. We use a two-phase optimization strategy, which addresses this problem. In the first phase, we search for the tree with the lowest total execution cost. In the second phase, we search for an optimal parallelization of the previously found tree. This approach may seem arguable, as some parallel trees are not considered at all. However, it is justified by at least three arguments [WFA95]: *(i)* we cannot assume that parallelism will to a large extent compensate for increased total amount of work *(ii)* the schedule with minimal total costs is likely to have small intermediate results, so that the additional processing introduced by Xchange operators will be small as well, *(iii)* two-phase optimization reduces the optimization time, by reducing the number of possible moves (it limits the search space).

The first phase of the two-phase optimization is done by standard query optimization and is performed by the Ingres DBMS. The second phase: finding a suitable parallelization for a given tree is the subject of this section.

5.1.1. Objective of Optimization

The general objective of query optimization is [GHK92], [vB89]:

Objective. *For a given query Q, a space of execution plans E, and a cost function that assigns a numeric cost to an execution plan $p \in E$, find the minimum cost execution plan that computes Q.*

In multiprocessor machines a higher degree of parallelism decreases the execution time. However, the total processing cost is increased as additional control and communication overhead is introduced. Extra work can be traded for reduced response time, but this cannot be done at any expense. Hence, the optimization problem involves a tradeoff between the execution time and the total processing cost.

In practice, we consider two conditions that constrain throughput degradation or cost-benefit ratio [GHK92]. Let the work and response time costs of the optimal-work plan be W_o and To and W_p and T_p for a considered plan respectively.

- System's throughput cannot degrade more than by a factor of k (a configurable parameter) i.e. W_p has to be lower than $k * W_o$.

- The ratio of the decrease in the response time to additional work required has to be lower than k i.e. $\frac{T_o - T_p}{W_p - W_o} < k$.

Choosing an appropriate plan at runtime is a difficult task that requires knowledge about the effects of the increased total processing cost and the reduced execution time on the query response time and system throughput. In order to obtain this knowledge, simulation experiments under several load conditions and using query execution plans with differing costs and execution times should be performed.

5.1.2. Elements of optimization

Finding an optimal query execution plan is a NP-hard optimization problem. Procedures that find the best solution operate on:

- A *state* which is an object of the search space (described in section 5.2).
- A set of *transformations* for changing states in the search space (described in section 5.3).
- A *search strategy* that chooses the next rule to be applied (described in section 6.4).
- A *cost function* which evaluates the computed solutions (described in section 6.5).

5.2. A state in a search space

We define a set of transformations for a query execution tree. We denote this rewriter's rule by $\mathbb{P}$ and introduce the following notation.

$$\mathbb{P}^{T}_{K,S,C}(X)|L$$

The $\mathbb{P}$ rule rewrites a given subtree (defined by X operator) and takes five parameters: L, T, K, S and C. A state in a search space is defined by the $< X, L, T, K, S, C >$ tuple. In the next paragraphs, we describe those parameters in more detail.

T parameter denotes the number of threads that can work in parallel. It is initially set to the maximal allowed level of parallelism (which can vary in dynamically changing environment).

If the $\mathbb{P}$ rule decides about parallelization of a given operator, it usually decreases the number of threads that are allowed to be created in the operator's subtree. However, we can exploit properties of some operators in order to use this parameter more wisely. Fully materializing operators (like Aggregation or HashJoin) are especially interesting.

For the Aggregation operator, before producing any output, the whole input has to be stored inside hash tables. As a result, operators working below and above Aggregation in a QET never work in parallel.

The HashJoin operator executes and fully materializes its right side input before starting to execute its left side. Because of that, creating T threads on both left and right side, will not exceed the limit for the overall number of threads working in parallel.

In transformation rules we also use Z variable to denote a possibility of creating any number (usually greater than one and not greater than T) of threads allowed in a child operator.

L **parameter** represents the number of streams the $\mathbb{P}$ rule has to produce.

The whole query execution tree has to produce a single stream. We use the family of Xchg operators to control the number of streams each operator is working on.

K **parameter** is used to denote if and how the output streams have to be partitioned. It is a (possibly empty) set of column names (attributes). A set of tuples is partitioned (i.e. each tuple is given an identifier defining its partition) if:

$\forall_{i,j}(\forall_{c\in K}\Pi_c(a_i) = \Pi_c(a_j)) \to Part(a_i) = Part(a_j)$

i.e. if any two tuples (a_i and a_j) equal on a the columns from K ($\Pi_c(a_i) = \Pi_c(a_j)$) are given the same partition identifier $Part(a_i) = Part(a_j)$.

If K is an empty set, streams do not have to be partitioned. Otherwise, for $K \neq \emptyset$, any two tuples from two different streams are different on columns from K. Additionally, since the property of being partitioned is only valid for multiple streams, $K \neq \emptyset$ implies $L > 1$.

Corollary 5.2.1. *If data is partitioned on K and $L \subseteq K$ then it is also partitioned on L.*

C **parameter** enforces $\mathbb{P}$ the rule to keep the stream clustered i.e.:

$\forall_{c\in C,i,j}\Pi_c(a_i) = \Pi_c(a_j) \to \forall_{i\leq k\leq j}\ \Pi_c(a_k) = \Pi_c(a_i)$

Uniquely clustered stream. If a stream S is both clustered and partitioned we say that S is *uniquely clustered* on $C' = C \cap K$ i.e.:

(i) its sequence of data is clustered on C' (i.e. tuples have to match only on columns from C')

(ii) it has a disjoint set of tuples with any other stream (S') with respect to C (i.e. $\Pi_{C'}(S) \cup \Pi_{C'}(S') = \emptyset$)

In practice we only encounter $C = K$.

S **parameter** enforces the $\mathbb{P}$ rule to keep streams of data sorted on columns from S. Operators that base on an assumption that their input is sorted on some set of columns set S to a list of these columns. This parameter assures that the sortedness will not be violated after applying any transformation.

5.3. Transformations

For each operator and requirement (defined by the parameters) we present a set of transformations. Only those transformations that may reduce execution time are presented.

Firstly, we focus on transformations that operate on unsorted data. For those operators we do not focus on clustering either. This is why we abbreviate $\mathbb{P}_{K,C,S}$ notation to $\mathbb{P}_K$. Also, we put $\mathbb{P}_{K(grpby)}(X)$ if $K = grpby$ and abbreviate $\mathbb{P}_{K(\emptyset)}(X)$ to $\mathbb{P}(X)$.

5.3.1. Aggregation

For transformation (5.3.1.1) we simply delegate the parallelization of the tree to the child operator.

Requirements: $L = 1$

$$\begin{array}{c} \mathbb{P}^T(\text{Aggr}_{grpby,aggr})|1 \\ | \\ \text{A} \end{array} \quad \rightarrow \quad \begin{array}{c} \text{Aggr}_{grpby,aggr} \\ | \\ \mathbb{P}^T(\text{A})|1 \end{array} \tag{5.3.1.1}$$

In transformation (5.3.1.2) we make the Aggregation operator parallel by enforcing it to produce multiple streams. Effectively, we reduce the problem to a different one. Streams have to be unified (as $\mathbb{P}$ has to produce a single stream), by XchgUnion operator. Finally, we assert correctness by adding an additional Aggregation operator with a modified list of aggregates $aggr'$ (see section 3.3). The T parameter is reset to $T - Z$ as we have already created Z threads.

Requirements: $L = 1 \wedge T \geq 2 \wedge S = \emptyset$

$$\begin{array}{c} \mathbb{P}^T(\text{Aggr}_{grpby,aggr})|1 \\ | \\ \text{A} \\ \\ \\ \end{array} \quad \rightarrow \quad \begin{array}{c} \text{Aggr}_{grpby,aggr'}|1 \\ | \\ \text{XU(Z)}|1 \\ | \\ \mathbb{P}^{T-Z}(\text{Aggr}_{grpby,aggr})|Z \end{array} \tag{5.3.1.2}$$

In transformation (5.3.1.3) T input streams are partitioned by the *grpby* columns as a result an additional Aggregation operator is not needed.

Requirements: $L = 1 \wedge T \geq 2 \wedge S = \emptyset \wedge grpby \neq \emptyset$

$$\begin{array}{c} \mathbb{P}^T(\text{Aggr}_{grpby,aggr})|1 \\ | \\ \text{A} \\ \\ \\ \end{array} \quad \rightarrow \quad \begin{array}{c} \text{XU(Z)}|1 \\ | \\ \text{Aggr}_{grpby,aggr}|Z \\ | \\ \mathbb{P}^{T-Z}_{K(grpby)}(\text{A})|Z \end{array} \quad \text{for } 2 \leq Z \leq T \tag{5.3.1.3}$$

In transformation (5.3.1.4), for $L \neq Z$ Xchg operator is required to produce the correct number of streams. In a special case when $L = Z$ we avoid adding an Xchange operator.

Requirements: $L > 1 \wedge K \neq \emptyset \wedge grpby \neq \emptyset \wedge K \subseteq grpby$

$$\begin{array}{c} \mathbb{P}^T_K(\text{Aggr}_{grpby,aggr})|L \\ | \\ \text{A} \end{array} \quad \rightarrow \quad \left\{ \begin{array}{ll} \begin{array}{c} \text{XCHG(Z:L)}|L \\ | \\ \text{Aggr}_{grpby,aggr}|Z \\ | \\ \mathbb{P}^{T-Z}_{grpby}(\text{A})|Z \end{array} & \text{for } \wedge 2 \leq Z \leq T + L \wedge Z \neq L \\ \\ \begin{array}{c} \text{Aggr}_{grpby,aggr}|L \\ | \\ \mathbb{P}^{T-L}_{grpby}(\text{A})|L \end{array} & \text{for } \wedge 2 \leq Z \leq T + L \wedge Z = L \end{array} \right. \tag{5.3.1.4}$$

Transformation (5.3.1.5) resolves the requirement to produce L streams partitioned on K.

Requirements: $K \neq \emptyset$

$$\begin{tikzcd}
\mathbb{P}_K^T(\text{Aggr}_{grpby,aggr})|L \arrow[r, phantom, "\rightarrow"] & \text{XHS}_K(\text{Z:L})|L \arrow[d, dash] \\
& \mathbb{P}_\emptyset^{T+L-Z}(\text{Aggr}_{grpby,aggr})|Z
\end{tikzcd} \tag{5.3.1.5}$$

5.3.2. Select

For the Select, operator parallelization can be either delagated to its child (transformation (5.3.2.1)) or, for partitioned streams, the XchgHashSplit operator can be used (transformation (5.3.2.2)).

$$\begin{tikzcd}
\mathbb{P}_K^T(\text{Select})|L \arrow[d, dash] \arrow[r, phantom, "\rightarrow"] & \text{Select}|L \arrow[d, dash] \\
\text{A} & \mathbb{P}_K^T(\text{A})|L
\end{tikzcd} \tag{5.3.2.1}$$

Requirements: $L > 1 \wedge K \neq \emptyset$

$$\begin{tikzcd}
\mathbb{P}_K^T(\text{Select})|L \arrow[d, dash] \arrow[r, phantom, "\rightarrow"] & \text{XHS}_K(\text{Z:L}) \arrow[d, dash] \\
\text{A} & \text{Select}|Z \arrow[d, dash] \\
& \mathbb{P}_{K(\emptyset)}^{T-Z}(\text{A})|L
\end{tikzcd} \tag{5.3.2.2}$$

For streams that have to be partitioned, decision about which transforation to use is an interesting question from a perfomance point of view. On the one hand, postponing partitioning the data enables Select operators to work in parallel. On the other hand, partitioning is enforced on a larger volume of data.

5.3.3. Project

The Project operator is especially interesting if forced to produce partitioned streams. If streams do not have to be partitioned, parallelization is delegated to the Project's child (transformation (5.3.3.1)).

Requirements: $K = \emptyset$

$$\begin{tikzcd}
\mathbb{P}_{K(\emptyset)}^T(\text{Project})|L \arrow[d, dash] \arrow[r, phantom, "\rightarrow"] & \text{Project}|L \arrow[d, dash] \\
\text{A} & \mathbb{P}_{K(\emptyset)}^T(\text{A})|L
\end{tikzcd} \tag{5.3.3.1}$$

If $K \neq \emptyset$ we can always use the (5.3.3.2) transformation.

Requirements: $K \neq \emptyset$

$$\begin{tikzcd}
\mathbb{P}_K^T(\text{Project}_{projlist})|L \arrow[d, dash] \arrow[r, phantom, "\rightarrow"] & \text{XHS}_K(\text{Z:L}) \arrow[d, dash] \\
\text{A} & \text{Project}_{projlist}|Z \arrow[d, dash] \\
& \mathbb{P}_{K(\emptyset)}^{T-Z}(\text{A})|Z
\end{tikzcd} \tag{5.3.3.2}$$

Algorithm 4 Check the applicability of transformation (5.3.3.3)
Returns $DEPS$ - a new list of columns for partitioning the data.

```
1: $DEPS \leftarrow \emptyset$ // A set of columns K depends on
2: for $p$ in reverse($projlist$) do
3:     if $p$ is of a form "$k =< expr >$" for any $k \in K \cup DEPS$ then
4:         $C \leftarrow$ columns used in $< expr >$
5:         $DEPS \leftarrow DEPS \setminus \{k\} \cup C$
6:         $K \leftarrow K \setminus \{k\}$
7: return $DEPS$
```

Additionally, if we update the set of columns that is used in partitioning, transformation (5.3.3.3) is applicable.

$$\begin{tikzcd} \mathbb{P}^T_K(\text{Project}_{projlist}|L) \arrow[d, dash] & \rightarrow & \text{Project}_{projlist}|L \arrow[d, dash] \\ \text{A} & & \mathbb{P}^T_{K'}(\text{A})|L \end{tikzcd} \qquad (5.3.3.3)$$

In this transformation, we investigate the projection list ($projlist$) of the Project operator.

Additionally, each projection can use column identifiers defined previously in the projection list. Algorithm 4 computes a set of columns that are used to compute columns from K. Moreover, this set is the smallest possible, which is important to reduce the execution cost.

5.3.4. Scan

The Scan operator can either split the data statically or dynamically (using the XchgDynamicSplit or the XchgDynamicHashSplit operators).

Requirements: $K = \emptyset \wedge C = \emptyset$

$$\mathbb{P}^T_{K,C}(\text{Scan})|L \quad \rightarrow \quad \text{Scan}|L(\text{Data divided statically}) \qquad (5.3.4.1)$$

$$\begin{tikzcd} \mathbb{P}^T_{K,C}(\text{Scan})|L & \rightarrow & \text{XDS(L)}|L \arrow[d, dash] \\ & & \text{Scan}|1 \end{tikzcd} \qquad (5.3.4.2)$$

Requirements: $K \neq \emptyset \wedge C = \emptyset$

$$\begin{tikzcd} \mathbb{P}^T_K(\text{Scan})|L & \rightarrow & \text{XDHS}_K(\text{L})|L \arrow[d, dash] \\ & & \text{Scan}|1 \end{tikzcd} \qquad (5.3.4.3)$$

The parallelization when $C \neq \emptyset$ uses ClusterTrees and join-range indices, which are described in section 6.6.

In practice, while searching for the optimal parallel QET we always favour static scan to dynamic partitioning. In the basic implemenation of the dynamic operators Static partitioning performs significantly better when comparing to our, *basic*, implemenation of the dynamic operators. The better implementation has been postponed for the future work (sec. 9).

5.3.5. HashJoin

Let K_1 and K_2 be sets of columns used for the join condition in the left and right respectively. Let $K_1' \subseteq K_1$ and $K_2' \subseteq K_2$.

For a single stream we can always delegate the parallelization to child operators (transformation (5.3.5.1)) or use XchgUnion operator and force HashJoin to produce multiple streams (transformation (5.3.5.2)).

Requirements: $L = 1$

$$\begin{tikzcd}[column sep=small]
& \mathbb{P}^T(\text{HashJoin})|1 \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{HashJoin}|1 \arrow[dl, dash] \arrow[dr, dash] & \\
\text{A} & & \text{B} & & \mathbb{P}^T(\text{A})|1 & & \mathbb{P}^T(\text{B})|1
\end{tikzcd} \tag{5.3.5.1}$$

In transformation (5.3.5.2) we reduce the problem to a different one.

Requirements: $L = 1 \wedge T > 1 \wedge S \neq \emptyset$

$$\begin{tikzcd}
\mathbb{P}^T(\text{HashJoin})|1 & \rightarrow & \text{XU(Z)}|1 \arrow[d, dash] \\
& & \mathbb{P}^T(\text{HashJoin})|Z
\end{tikzcd} \tag{5.3.5.2}$$

If the HashJoin operator is forced to produce multiple streams, its left and right children have to produce multiple streams. We especially focus on parallelizing HashJoin's left side, which is assumed to be more profitable (as building the hash table from the smaller input is faster), although a symmetrical transformation is also valid.

In order to preserve correctness, we can either duplicate right streams using the XchgBroadcast operator, or enforce these streams to be partitioned on the key columns.

In the former case (transformation (5.3.5.3)), each HashJoin operates on the same output from the right child. In this scenario, each operator duplicates the work of creating its own instance of hash table significantly increasing, the total execution cost [*].

Requirements: $L > 1$

$$\begin{tikzcd}[column sep=small]
& \mathbb{P}^T_K(\text{HashJoin})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{HashJoin}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\text{A} & & \text{B} & & \mathbb{P}^{T+L}_K(\text{A})|L & & \text{XBC(L)}|L \arrow[d, dash] \\
& & & & & & \mathbb{P}^{T+L}_{K(\emptyset)}(\text{B})|1
\end{tikzcd} \tag{5.3.5.3}$$

In the latter case (transformation (5.3.5.4)), unique input is split between all HashJoin operators. It is a correct transformation since, each operator receives all and only those tuples for which a hash function returns the same value on columns from K_1 or K_2.

[*]Computing this table only once and distributing it between all HashJoin operators for read-only access is discussed in sec. 9.

$$\begin{tikzcd}[column sep=tiny] & \mathbb{P}^T_\emptyset(\text{HashJoin})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{HashJoin}|L \arrow[dl, dash] \arrow[dr, dash] & \\ \text{A} & & \text{B} & & \mathbb{P}^T_{K(K_1)}(\text{A})|Z & & \mathbb{P}^T_{K(K_2)}(\text{B})|L \end{tikzcd} \tag{5.3.5.4}$$

Multiple, partitioned streams. If the HashJoin operator is required to produce multiple, partitioned streams many different transformations are available.

The simplest way of meeting such requirements is to use XchgHashSplit operator as presented in (5.3.5.5) transformation.

$$\begin{tikzcd} \mathbb{P}^T_K(\text{HashJoin})|L & \text{XHS}_K(\text{Z:L})|L \arrow[d, dash] \\ & \mathbb{P}^{T+L-Z}_{K(\emptyset)}(\text{HashJoin})|Z \end{tikzcd} \tag{5.3.5.5}$$

Transformation (5.3.5.6) does not add the XchgHashSplit operator, but tries to delegate data partitioning to child operators. This is not always possible. If the HashJoin operator forces its left and right child to produce data partitioned on K, then the output data is also partitioned on K (as required). However, in order to ensure correctness, K has to be *compatibile* with the join condition, meaning that, the data has to also be partitioned on K_1 and K_2.

$$\begin{tikzcd}[column sep=tiny] & \mathbb{P}^L_K(\text{HashJoin})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{HashJoin}|L \arrow[dl, dash] \arrow[dr, dash] & \\ \text{A} & & \text{B} & & \mathbb{P}^T_{K(K_1')}(\text{A})|L & & \mathbb{P}^T_{K(K_2')}(\text{B})|L \end{tikzcd} \tag{5.3.5.6}$$

Correctness of transformation (5.3.5.6)**.** If $\exists_{c \in K} c \notin K_1 \cup K_2$ (K has a column that is not present in the join condition), we do not allow this transformation. Clearly, column c may not exists in both A and B subtrees (but certainly c belongs to at least one of those). Let us assume that $c \notin Columns(A)$. The transformation can still be correct if c is a primary (or foreign) key column that has its corresponding column in B. However, if it was the case, then most probably c would have been mentioned in the join condition (which breaks our initial assumption).

Because of that, we simply state that transformation (5.3.5.6) may be applied if

$$K \subseteq K_1 \cup K_2$$

Using corollary 5.2.1 we put $K_1' = K_1$ and $K_2' = K_2$.

Deadlock hazard. Transformations 5.3.5.4 and 5.3.5.6 may, under certain circumstances, lead to a deadlock. Xchange operators assume that all their parents will sooner or later consume data prepared for them and, as a result, change full buffers into empty. However, for the HashJoin operator we may decide not to evaluate its left child if its right child returns no data (as the result of this join operation is known beforehand - an empty set of tuples) we result in a deadlock.

If XchgHashSplit operator tries to output two streams of data, one of which is consumed and one is not, buffers get clogged.

In figure 5.3.1 the XchgHashSplit operators produce two streams for the HashJoin operators (HJ and HJ'). Let's assume that the right XHS operator returns no data (dashed line). Because of the optimization mentioned before, the HashJoin (HJ') operator decides not to evaluate its left

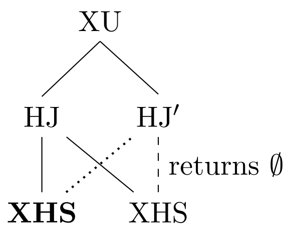


Figure 5.3.1: Deadlock condition.

child (dotted line). However, the left HashJoin operator is still evaluating its left child (the **XHS** operator). The XchgHashSplit operator produces data both for left and right HashJoin. The non consumed data (prepared for HJ') eventually fills all the buffers. Producers of the XchgHashSplit operator cannot obtain an empty buffer. We result in a deadlock.

5.3.6. MergeJoin

The MergeJoin operator requires the input streams to be sorted. In the family of Xchg operators only XchgBroadcast and XchgFork preserve this property. Therefore, if any other Xchg operator is used below MergeJoin on its left or right side, the data has to be explicitly sorted.

The Sort operator has two major disadvantages of being computationally expensive and materializing the whole input. As a result, we do not consider transformations introducing the Sort operator as promising.

Transformations that do not require the Sort operator broadcast either the left or right input introducing data partitioning on the second child. Alternatively, we can enforce children operators to produce uniquely clustered, sorted streams.

Replacing MergeJoin with HashJoin In our transformation rules we do not change existing operators to different ones. However, strict limitations in dealing with MergeJoin make replacing this operator with HashJoin feasible. A simple benefit of using HashJoin is a much wider set of transformations available.

Single, no-partitioned stream. For a single, no-partitioned stream we delegate reduce the problem to a different one (transformation (5.3.6.1)).

$$\begin{tikzcd}[row sep=small] \mathbb{P}^T(\mathrm{MJ})|1 \arrow[r, phantom, "\rightarrow"] & \mathrm{XU(T)}|1 \arrow[d, no head] \\ & \mathbb{P}^T(\mathrm{MJ})|T \end{tikzcd} \tag{5.3.6.1}$$

Multiple, no-partitioned streams. Transformation (5.3.6.2) does not parallelize B subtree as all transformations would require the Sort operator.

$$\begin{tikzcd}
 & \mathbb{P}^T(\mathrm{MJ})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \mathrm{MJ}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{A} & & \mathrm{B} & & \mathbb{P}^T_S(\mathrm{A})|L & & \mathrm{XBC(L)}|L \arrow[d, dash] \\
 & & & & & & \mathrm{B}
\end{tikzcd} \tag{5.3.6.2}$$

Transformation (5.3.6.3) requires both flows to produce uniquely clustered sorted streams. Those requirements enforced on child operators assert correct output streams provided that two corresponding input streams (joined to produce one output stream) are clustered in the same manner.

$$\begin{tikzcd}
 & \mathbb{P}^T(\mathrm{MJ})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \mathrm{MJ}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{A} & & \mathrm{B} & & \mathbb{P}^T_{K,S,C}(\mathrm{A})|L & & \mathbb{P}^T_{K,S,C}(\mathrm{B})|L
\end{tikzcd} \tag{5.3.6.3}$$

Multiple, partitioned streams. Again, the simplest solution is to use XchgHashSplit operator and reduce the problem to a transformation for a no-partitioned stream.

$$\begin{tikzcd}
\mathbb{P}^T_K(\mathrm{MJ})|L & \rightarrow & \mathrm{XHS}_K(Z : L)|L \arrow[d, dash] \\
 & & \mathbb{P}^T_{K(\emptyset)}(\mathrm{MJ})|Z
\end{tikzcd} \tag{5.3.6.4}$$

Another solution delegates the requirement to produce sorted, partitioned streams to one of MergeJoin's children. This is a correct transformation since the data partitioning is done only in one child.

$$\begin{tikzcd}
 & \mathbb{P}^T_K(\mathrm{MJ})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \mathrm{MJ}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{A} & & \mathrm{B} & & \mathbb{P}^T_{K,S}(\mathrm{A})|L & & \mathrm{XBC(L)}|L \arrow[d, dash] \\
 & & & & & & \mathrm{B}
\end{tikzcd} \tag{5.3.6.5}$$

Multiple, uniquely clustered sorted streams. The most complicated situation happens when MergeJoin operator is required to produce multiple uniquely sorted streams.

The simplest solution is to behave similarly to HashJoin and parallelize only this child that operates on columns from C.

$$\begin{tikzcd}
 & \mathbb{P}^T_{K(\emptyset),S,C}(\mathrm{MJ})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \mathrm{MJ}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{A} & & \mathrm{B} & & \mathbb{P}^T_{K(\emptyset),S,C}(\mathrm{A})|L & & \mathrm{XBC(L)}|L \arrow[d, dash] \\
 & & & & & & \mathrm{B}
\end{tikzcd} \tag{5.3.6.6}$$

5.3.7. TopN

TopN is used mainly as the topmost operator. It is commutative (see section 3.3) so we can easily divide its computation between multiple threads with a cost of additional TopN operator.

Transformation (5.3.7.1) presents this operation.

$$\begin{tikzcd}
\mathbb{P}^T\text{TopN}|1 \arrow[d, dash] & \rightarrow & \text{TopN}|1 \arrow[d, dash] \\
\text{A} & & \text{XU(Z)}|1 \arrow[d, dash] \\
& & \mathbb{P}^{T-Z}(\text{A})|Z
\end{tikzcd} \tag{5.3.7.1}$$

5.3.8. Ordered Aggregation

Ordered Aggregation requires the input to be clustered. Therefore, transformations on this operator set the C parameter in the $\mathbb{P}$ rule. We assume that the *groupby* set of columns is non-empty (as otherwise the OrderedAggregation is indistinguishable from normal Aggregation).

A single stream.

For transformation (5.3.8.1) we simply delegate parallelization of the tree to the child operator.

$$\begin{tikzcd}
\mathbb{P}^T_{K(\emptyset)}(\text{OrdAggr}_{grpby,aggr})|1 \arrow[d, dash] & \rightarrow & \text{OrdAggr}_{grpby,aggr} \arrow[d, dash] \\
\text{A} & & \mathbb{P}^T_{K(\emptyset),C(grpby)}(\text{A})|1
\end{tikzcd} \tag{5.3.8.1}$$

Transformation (5.3.8.2) is equivalent to (5.3.1.2). As the input streams to the bottom OrdAggr operator may not be uniquely clustered the output of the XchgUnion operator may be incorrect. Hence we have to add an additional Aggregation operator (again we exploit the commutative property of Aggregation sec. 3.3). What is more, the child is enforced to produce clustered streams.

$$\begin{tikzcd}
\mathbb{P}^T_{K(\emptyset),C(\emptyset)}(\text{OrdAggr}_{grpby,aggr})|1 \arrow[d, dash] & \rightarrow & Aggr_{grpby,aggr'}|1 \arrow[d, dash] \\
\text{A} & & \text{XU(Z)}|1 \arrow[d, dash] \\
& & \text{OrdAggr}_{grpby,aggr}|Z \arrow[d, dash] \\
& & \mathbb{P}^{T-Z}_{K(\emptyset),C(grpby)}(\text{A})|Z
\end{tikzcd} \tag{5.3.8.2}$$

We can also rewrite the problem to a different one issuing transformation (5.3.8.3).

$$\begin{tikzcd}
\mathbb{P}^T_{K(\emptyset),C(\emptyset)}(\text{OrdAggr}_{grpby,aggr})|1 \arrow[d, dash] & \rightarrow & \text{XU(Z)}|1 \arrow[d, dash] \\
\text{A} & & \mathbb{P}^{T-Z}_{K(\emptyset),C(\emptyset)}(\text{OrdAggr}_{grpby,aggr})|Z \arrow[d, dash] \\
& & \text{A}
\end{tikzcd} \tag{5.3.8.3}$$

Multiple, non-partitioned, non-clustered streams.

We avoid adding an additional Aggregation operator if the child operator is known to produce uniquely clustered streams. Then a given group of tuples reaches particular instance of OrderedAggregation enabling it produce the correct output.

$$\begin{array}{c} \mathbb{P}^{T}_{K(\emptyset),C(\emptyset)}(\text{OrdAggr}_{grpby,aggr})|L \\ | \\ \text{A} \end{array} \quad \rightarrow \quad \begin{array}{c} \text{OrdAggr}_{grpby,aggr}|L \\ | \\ \mathbb{P}^{T-Z}_{K(grpby),C(grpby)}(\text{A})|L \end{array} \tag{5.3.8.4}$$

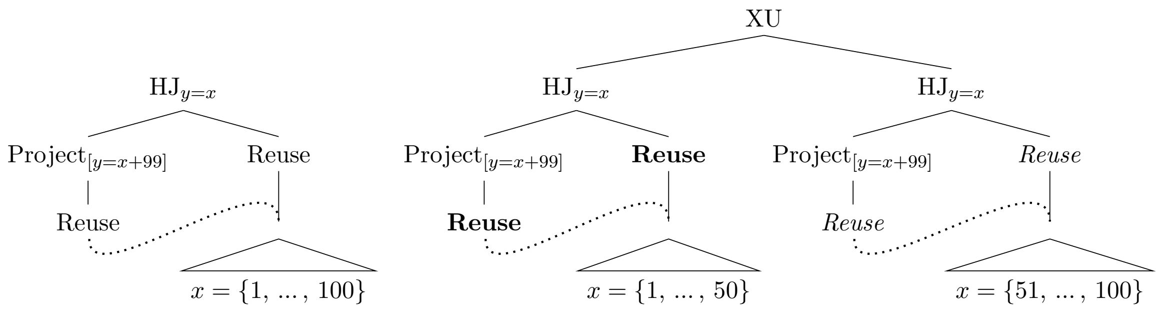
Multiple, partitioned or clustered streams.

If OrderedAggregation is forced to produce partitioned (on K) or clustered (on C, which equals K) streams, then we have to take a look at its *grpby* set of columns. If K (or *grpby*) functionally depends on *grpby* (or K) then it is safe to delegate the transformation.

5.3.9. Reuse

The Reuse operator is inevitably connected with binary, especially join, operators. Assuring validity of transformations parallelizing the Reuse operator is an non-trivial task. First of all, if one of the Reuse operator is parallelized its counterpart also has to be parallelized (as the number of streams to both operators has to the same).

Figure 5.3.2a presents a part of a sequential plan, whereas 5.3.2b presents a parallel version of this plan with statically partitioned data. The sequential plan returns one tuple (its value equals to 100), whereas the parallel plan does not return any tuples (the HashJoin operators do not find any matching tuples).



(a) Sequential plan. (b) Invalid parallel plan.

Figure 5.3.2: Problems with parallelizing the Reuse operator

Transformations of the Reuse operator require performing an additional check. We remembering the list of keys used in the join operator. If, on our way down to any of the Reuse operators, we had noticed that any of the keys had been altered (e.g. by the Aggregation operator or, as in the example from figure 5.3.2b, the Project operator), then the parallelization is invalid.

Queries from the TPC-H benchmark, which used the Reuse operator (see section 7.1), could not have been parallelized in this manner (there is always Project or Aggregation operator on a path between the Reuse and a join operator that affects the a key used in the join predicate).

5.3.10. Transformations with S parameter.

If the $\mathbb{P}$ rule has the S parameter set, we will not use any of the Xchg operators except for XchgBroadcast and XchgFork, which are the only operators that do not change the sortedness property of a stream.

XchgBroadcast operator may only be present for joining operators. Transformation 5.3.10.1 is valid both for MergeJoin and for HashJoin. Also, a symmetrical transformation with the XchgBroadcast operator in the left subtree exists.

$$\begin{tikzcd}
& \mathbb{P}^T_{K,C,S}(\text{Join})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{Join}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\text{A} & & \text{B} & & \mathbb{P}^T_{K,C,S}(\text{A})|L & & \text{XBC(L)}|L \arrow[d, dash] \\
& & & & & & \text{B}
\end{tikzcd} \tag{5.3.10.1}$$

Transformation (5.3.10.2) makes both children of a Join operator work in parallel. Thanks to that, the access to data from the left subtree is reduced. Also this transformation may be used

$$\begin{tikzcd}
& \mathbb{P}^T_{K,C,S}(\text{Join})|L \arrow[dl, dash] \arrow[dr, dash] & & \rightarrow & & \text{Join}|L \arrow[dl, dash] \arrow[dr, dash] & \\
\text{A} & & \text{B} & & \text{XF(id)}|L \arrow[d, dash] & & \text{XF(id)}|L \arrow[d, dash] \\
& & & & \mathbb{P}^T_{K,C,S}(\text{A})|L & & \mathbb{P}^T_{K,C,S}(\text{B})|L
\end{tikzcd} \tag{5.3.10.2}$$

Chapter 6

Cost model

A cost model is an essential component in query optimization. It predicts a cost of an execution plan allowing us to choose supposedly the best.

In this section, we assume that for a given operator, the type of its input (e.g. strings, integers) and estimates about its cardinalities we are able, to some extent, to estimate the response time. The response time can be measured in seconds but also in the number of CPU cycles. In practice this is a non-trivial task, since a function providing cost estimates is multi-dimensional. The most obvious correlation is between the response time and the size of data to be processed. However, cache influence (its size, hierarchy), memory access patterns, processor capabilities, other queries currently executed definitely cannot be neglected.

For the purposes of this thesis, a simple model for cost estimates was implemented. This implementation depended mainly on estimated cardinalities and the type of the data.

6.1. Estimating response time.

In this section, we present a cost model that represents the tradeoffs amongst the aspects of parallel execution, which are presented in [GHK92]. We adapt this example to our execution model and express it entirely using Xchange operators.

A concept of *time descriptor.*

For a given operator tree P, we define *time descriptor* $t = (p_f, p_l)$, where p_f (resp. p_l) is the estimated time of producing the first (last) tuple by P. Time descriptor incorporates information about data dependencies in P.

Property. *If S is the set of all subtrees in P and S' the minimal subset of S that has to be finished before P outputs the first tuple, then $\forall_{T \in S'} p_f \geq s'_l$.*

This means that the value of t_f cannot be lower than the maximum t_l of all subtrees of P that materialize its flow. We call S' *materialized front* of P.

6.1.1. Resource contention free execution

Firstly, we focus on estimating response times in a contention free environment. Unary operators may execute in either pipelined or materialized execution schema. For both composition methods, we derive formulae for the resulting time descriptor t.

Unary operators.

For the pipelined schema, we assume that time of producing the first tuple [*] is negligible. On the other hand, if an operator materializes its flow, we state that the time of producing the last tuple (t_l) equals the time of producing the first tuple (t_f). This is a reasonable approximation as the parent operator has an instant access to the materialized (and already processed) data.

Therefore, pipelined execution of two operators P and C (producer and consumer) is described by time descriptor $t = (t_f, t_l)$ equal to

$$(t_f, t_l) = (p_f + c_f, p_l + c_l) = (p_f, p_l + c_l)$$

If P is a materializing operator then

$$(t_f, t_l) = (p_l + c_l, p_l + c_l)$$

Binary operators.

For binary operators, as they have a different operation schema, we define resulting time descriptor t separately.

HashJoin. Let A and B be left and right children of the HashJoin operator. Also let T_{build} be the cost of creating the hash table and T_{probe} be estimated cost of the probing phase in which HashJoin operates in a pipelined schema (see 3.5). Then

$$(t_f, t_l) = b_l + T_{build} + (a_f, a_l + T_{probe})$$

where the addition to a scalar is defined by

$$a + (b, c) \equiv (b + a, c + a)$$

The operator firstly processes its right side (hence b_l), then materializes the flow in its hash table ($b_l + T_{build}$), and then it processes its left child starting from a_f and finishing at $a_l + T_{probe}$. Again we assumed that the cost of producing the first tuple is negligible.

MergeJoin operator's cost model is simpler since both left and right (A and B) children operate in a pipeline. If T_{merge} is merging cost, then

$$(t_f, t_l) = (a_f + b_f, a_l + b_l + T_{merge})$$

6.2. Estimates in parallel execution

An interesting situation in cost calculus is related to transformations derived from (5.3.10.2). In those situations XchgFork enables both left and right subtrees of a binary operator to operate in parallel. The HashJoin operator does not proceed with the left subtree until its hash table is built. Hence, we cannot benefit more than $min(b_l+T_{build}, a_f)$ that can be subtracted from the left subtree's execution time. The time descriptor is given by

$$(t_f, t_l) = (b_l + T_{build} + a_f, a_l - min(b_l + T_{build}, a_f) + T_{probe})$$

[*]from the VectorWise DBMS perspective this is also time of processing the first vector.

For MergeJoin operator, we save $min(a_f, b_f)$ at most. Hence

$$(t_f, t_l) = (a_f + b_f, a_l + b_l) + T_{merge} - min(a_f, b_f)$$

6.3. Resource contention

In the previous section, we assumed no resource contention. This allowed us to derive a simple cost calculus for a given operator tree. Now, we relax this assumption.

Apart from synchronization overhead, parallel execution introduces context changing overhead, memory access penalty (cache invalidation, memory access congestion). There are also other factors like data skew or computation skew, which affect our estimations of the response time.

Because of that, we introduce a parallel execution penalty, which is expressed by a scalar. The penalty differs depending on the type of execution. We consider two types of parallel execution *(i)* independent parallel execution (IPE) and *(ii)* dependant parallel execution (DPE). IPE is present between materialized frontiers of a given operator, whereas we assume DPE when referring to pipelined execution.

Both IPE and DPE suffer from having to share resources and data or computation skew, albeit only DPE has to synchronize. Because of that, we use a simplification and calculate the cost penalty only for dependent execution. We denote this penalty as $\gamma(k)$.

Now, without presenting how to obtain this value we adjust our cost calculus to incorporate resource contention. Also, in order to avoid exponential penalty propagation, the overhead is added only in transformations actually introducing parallelism and requiring synchronization i.e. for Xchange operators. Xchange operators work in a pipelined schema. Thus, if P is the child operator with time descriptor (p_f, p_l) and C is the Xchange operator, then

$$(t_f, t_l) = ((p_f + c_f),\ p_l + c_l + (p_l + c_l - p_f - c_f) * \gamma(k))$$

6.3.1. Cost penalty calculation

The purpose of introducing a time descriptor was to distinguish between synchronized and unsynchronized execution. Execution of materialized frontiers is done independently, whereas the pipelined execution requires synchronization from Xchange operators.

Synchronization overhead model. For each Xchange operator, given a time descriptor t we know that t_f time is spent in a subtree that finally gets materialized and does not require synchronization. Contrary, $t_l - t_f$ is spent in synchronized execution. For synchronized execution there are two extremes for the response time. Let us assume that the Xchange operator's subtree is split into L streams. The lowest response time happens if a query gets parallelized and its response time is close to $t_l - t_f$. The worst scenario is when subtrees assumed to execute in parallel are in fact executed sequentially. Then the response time is closer to $(t_l - t_f) * L$ extreme. The response time is a parameterized linear interpolation of the response time t', where $t' \in [(t_l - t_f), (t_l - t_f) * L]$;

Resource usage model. The usage of a specific resource is modeled by two parameters - its availability and the current demand for it. Ideally, when calculating the demand we also take time intervals into account. This would enable us to model hot spots as well as more accurately predict interference of other queries. However, for simplicity, we assume that resources occupied by other queries are kept occupied throughout the execution time of the query currently optimized.

Now for each i and resource r_i we calculate the load average by $max(1, \frac{demand(r_i)}{availability(r_i)})$, by which we will penalize the response time. For example, suppose that 2 other queries occupy three out of 4 cores. If the currently optimized plan requires 3 cores then the load average of CPU is $\frac{5}{4}$.

Penalty Given the above we define penalty $\gamma(k)$ as parameterized approximation:

$$\gamma(k) = 1 + k * \frac{t' - (t_l - t_f)}{(t_l - t_f) * L - (t_l - t_f)} * max_i(1, \frac{demand(r_i)}{availability(r_i)})$$

The parameter k in this formula has to be adjusted after performing benchmarks tests. Also a more accurate, non linear approximation of the response time may be used.

6.4. Search strategy

For the known set of transformations a given non-parallel query execution tree can be rewritten into a very large set of equivalent parallel trees. The search-space of all those trees grows exponentially with the number of operators. This is because for each operator and the set of parameters for $\mathbb{P}$ rule usually at least two transformations are valid.

An outline of a transformation based search algorithm of the optimal query execution plan is presented in [GCD$^+$91]. In our case, the algorithm searches for the optimal parallel plan starting from its optimal non-parallel counterpart.

6.4.1. Algorithm description

Branch-And-Prune approach Algorithm 5 describes a branch-and-prune approach for limiting the state-space of all possible trees. The initial cost limit is set to the estimated cost of a non-parallel QET. Later on, only those transformations that decrease the upper cost limit are considered.

It is therefore important to find the best plan as early as possible and let the algorithm prune all trees that are estimated to be not optimal.

In order to achieve that Algorithm 5 first tries more promising moves.

Pursuing only a selected few from the set of all possible moves is another heuristic that requires sorting the possible transformations by its promise. This has to be done beforehand by the programmer.

Dynamic programming Answers for previously computed states are stored and not recomputed.

6.5. Cost Function

For each operator we implemented a simple cost model, which estimates the number of CPU cycles required to process the whole input. We considered the following dimensions:

(i) the size of input data,

(ii) the size of output data,

(iii) the size of memory accessed,

Algorithm 5 RuleP(Op, T, L, K, S, C, Limit)
Find the best possible query execution plan

```
1: if $tuple < Op, T, L, K, S >$ is in the look-up table then
2:     $(plan, cost)$ = lookup($Op, T, L, K, S, C$)
3:     if $cost < Limit$ then
4:         return $(plan, cost)$
5:     else
6:         return $failure$
7:
8: // Current state has to be computed
9: $T$ = Transformations($Op, T, L, K, S, C$)
10: $Ts$ = SortByPromise($Op$, $T$)
11: $best\_cost = Inf$
12: for $t$ in $Ts$ do
13:     $extra\_cost$ = ExtraCost($t$)
14:     if $extra\_cost < Limit$ then
15:         $(plan, cost)$ = RuleP($Op, T, L, K, S, C$)
16:         if $cost < best\_cost$ then
17:             $best\_cost = cost$
18:             $best\_plan = plan$
19: if $best\_cost == Inf$ then
20:     return $failure$
21: else
22:     set $< Op, T, L, K, S, C >$ to $(best\_plan, best\_cost)$ in the look-up table
23:     return $(best\_plan, best\_cost)$
```

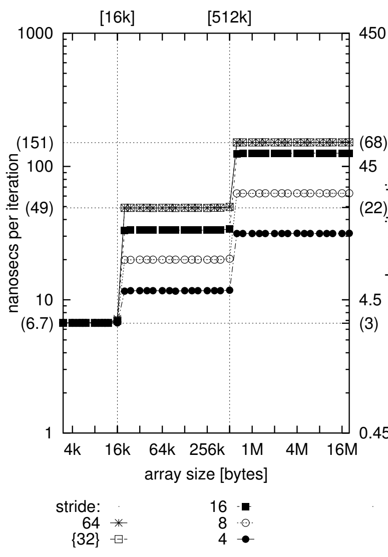
(iv) the type of data.

The size of the input or the output data cannot be predicted with 100% accuracy as not all the information can be obtained from histograms. For example we do not know how many tuples with a string attribute match an arbitrary regular expression (given in the LIKE clause). This is one of the arguments for creating query execution plan dynamically i.e. to determine the shape of the query execution tree while actually executing the query [GW89].

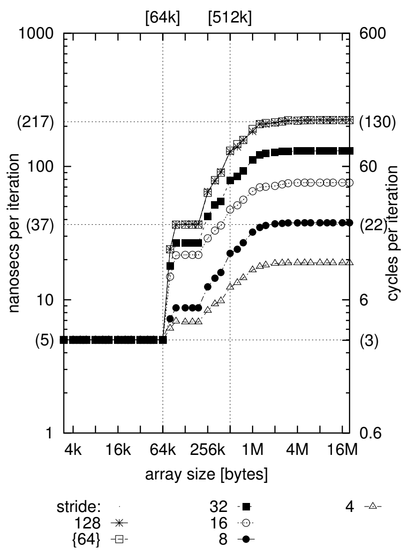
For the HashJoin and Aggregation operators, the size of the hash table affects the performance in a non-linear, CPU-dependent manner. Figure 6.5.1 presents the time required to iterate through an array of values. A *stride* is the difference in bytes between two consecutive memory accesses. The iteration walks "backward" through the memory so the processors prefetching (which is designed for "forward-oriented" memory access patterns) does not affect the experiment.

Results

Computed costs were sometimes very different (with errors up to a few hundred times) to the actual costs. The variance was especially high for hashing operators (Aggregation and HashJoin), as their operational model is especially complicated. However, in general, the implemented model was sufficient as for the purposes of this master thesis.



(a) Intel PC (Katmai 450)



(b) AMD (Athlon 600)

Figure 6.5.1: The impact of the processor caching. If the accessed array does not fit into faster cache the performance deteriorates in a non-linear, processor-specific manner. Courtesy of Stefan Manegold [Man02].

6.6. Parallelization using cluster trees.

This structure enables us to produce uniquely clustered, sorted streams. As a result, we are given a very fast way to resolve many transformation requirements enforced by the OrderedAggregation and the MergeJoin operators.

If the MergeJoin operator requires its children to produce multiple clustered and uniquely partitioned streams, then, finally, the request is passed on to Scan operators.

Let as assume that these Scan operators work on tables A and B, and that table A is large (i.e. it references B). Now if there exists a join-range index between A and B we issue the following algorithm:

(i) divide table A into L equal ranges R_A (L is the parameter of $\mathbb{P}$ operator)

(ii) for each range $r_A \in R_A$ map this range using the join-range index onto the corresponding range in table B obtaining r_B

(iii) remap range r_B back to table A obtaining r'_A

(iv) update all other ranges in R_A so that r'_A do not intersect with other ranges.

As not every range of the size of one L-th of the smaller table corresponds to one L-th of the larger table, in the mentioned algorithm we start with the larger table in order to reduce computation skew between different threads. This approach relies on a fact that we should only try to reduce relative differences between ranges in the larger table as those constitute the majority of the computation.

In the second step a range of one L-th of the larger table is mapped into the smaller table. The r_B range has already the property of being clustered and uniquely partitioned as it is obtained from a join-range index chunk (collateral 2.2.3). The corresponding range r'_A is obtained by mapping r_B

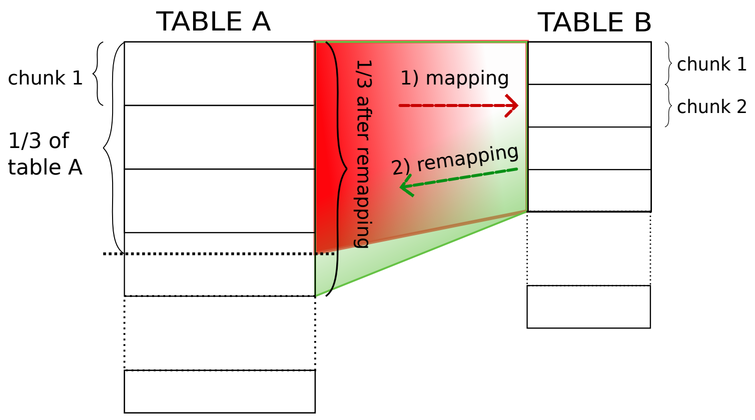


Figure 6.6.1: Obtaining uniquely clustered streams.

back to table A. Table r'_A is also clustered and uniquely partitioned with values matching those from range r_B.

The mapping process is an in-memory binary search performed on the join-range index. The whole mapping process is performed in time complexity of $O(L * log(c))$ where c is the number of chunks in the join-range index.

Figure 6.6.1 presents the application of the algorithm for one of the ranges in a situation where the L parameter is three.

This algorithm may result in data skew as computed ranges may differ in size.

Chapter 7

Results

7.1. TPC-H benchmark analysis

Our solution has been tested and optimized for the TPC-H benchmark, which is widely used in the database community as a yardstick to assess the performance of database management systems against large scale decision support applications. The benchmark is designed and maintained by the Transaction Processing Performance Council. This benchmark defines a data warehouse with customers placing orders consisting of multiple line items. Each line item consists of a number of parts; and parts are supplied by multiple suppliers. The benchmark comes with a data generator and set a of 22 reporting queries. The size of the database can be scaled using a scaling factor (SF). The default SF is 1, which represents 1 GB of data.

Speedup tests were performed on scale factor 100 (about 180GB of data). A Linux server machine, having two chip multiprocessors[*], each having four physical cores and the support for Symmetric Multi-Threading.

Figure 7.1.1 presents obtained results. For each query we chose the best of four runs in order to minimize the noise. All the queries from the benchmark were parallelized and run faster than in their sequential version.

General results are satisfactory. In the next sections, we present a brief description of each query with a parallel QET used to calculate the answer[†]. QET also include the percentage of the time spent in a given subtree, and the number of tuples processed passed between given operators. Values were gathered from parallel plans with the maximum parallelization level of two [‡]).

The results presented, were gathered at some point of the development process. An interesting fact is that if the benchmarks had been performed one month before, the presented results would be significantly worse. Optimizations (new rewriting rules, reducing the synchronization overhead etc.) allowed as to decrease the overall time to run all 22 TPC-H queries with 4 threads from 115.5 seconds to 89.5 seconds (an improvement of 22.5%). More improvements may result in even better outcome.

Query 1

This query obtained the best speedups (it runs 17.71 sec with one thread and 2.67 sec on 8 threads). It consists of a simple scan-select-aggregation path, processes a large amount of data (over 600 million tuples) and the output of the aggregation operator is very small (4 tuples). Those facts

[*]Intel(R) Xeon(R) X5560 2.80GHz

[†]For the SQL statements please refer to [TPC].

[‡]The maximum parallelization level is the initial value of T parameter passed to $\mathbb{P}$ rule. We may also say that query uses T threads.

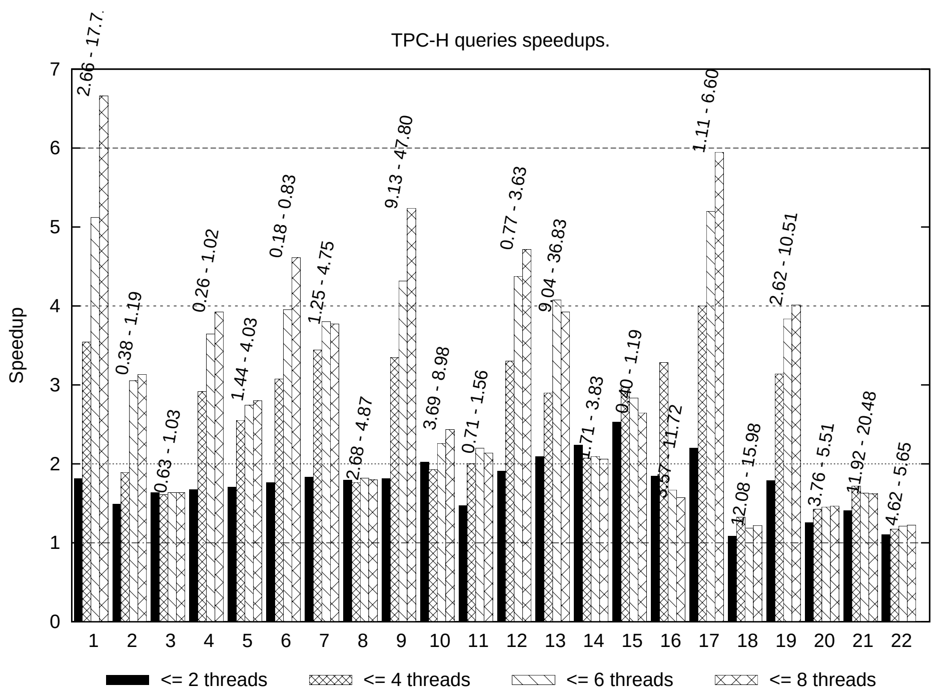
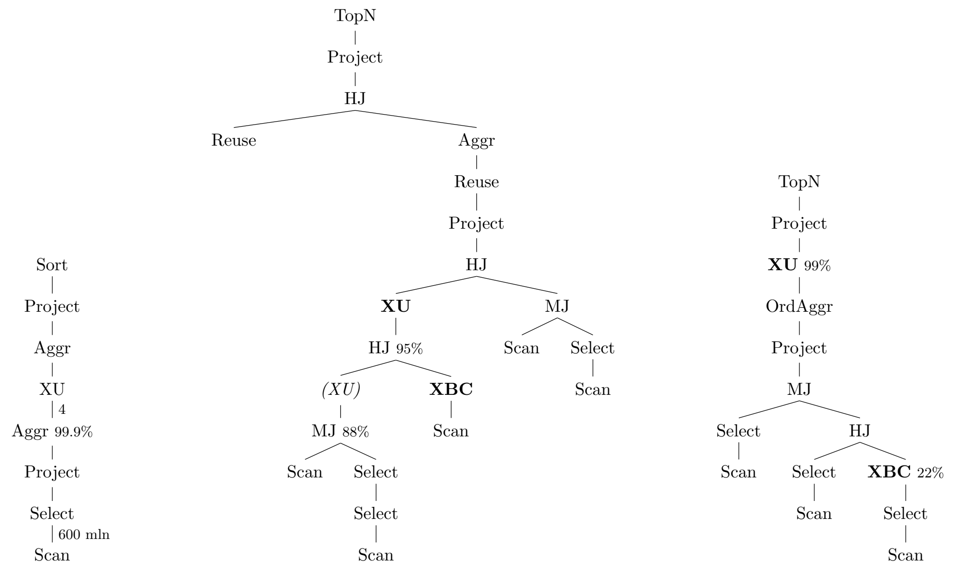


Figure 7.1.1: Speedups obtained on Linux sever machine (8 physical cores) on SF 100. Additionally, each query the fastest run and the slowest run in seconds is presented. For each parameter, the best of four runs was used.



(a) Query 1

(b) Query 2 - starting from 4 threads the parallel plan has been changed.

(c) Query 3

Figure 7.1.2: Parallel QET used in TPC-H benchmark.

make the parallelization very easy to detect and assure small additional overhead. Figure 7.1.2a presents this query after parallelization. An additional Aggregation operator was used. More than 99.9 % of computation time is spent in the subtree below the XchgUnion operator. As a result, nearly all of the computation is performed in parallel.

Query 2

Query 2 is a multi-join query that reuses common subexpressions (the Reuse operator). When the maximum parallelization level was set to 4, we ran a different plan (operators in bold were replaced by XchgUnion in italics). The reason for doing so, was a higher total cost required to create multiple hash tables and the cost of building the XchgBroadcast operator with more buffers. This was a reasonable decision as the operator build phase is relatively costly for this fast query. For two threads the building phase took about 10% of the whole query. By changing the plan, starting with 4 threads, this cost was reduced to about 6%.

Query 3

In the third query, we are able to put the XchgUnion operator near the root of the QET. As a result, 99% minus 22% (as we use XchgBroadcast) of the tree gets parallelized. Because of the relatively low total cost of this query (sequential plan finishes in about 1 second) the optimizer decides to limit the parallelization level to two. The query uses a join-range index to split the ranges in MergeJoin's left side input, and HashJoin's left side input. This is possible since HashJoin does not change the ordering of tuples from its outer input.

Query 4

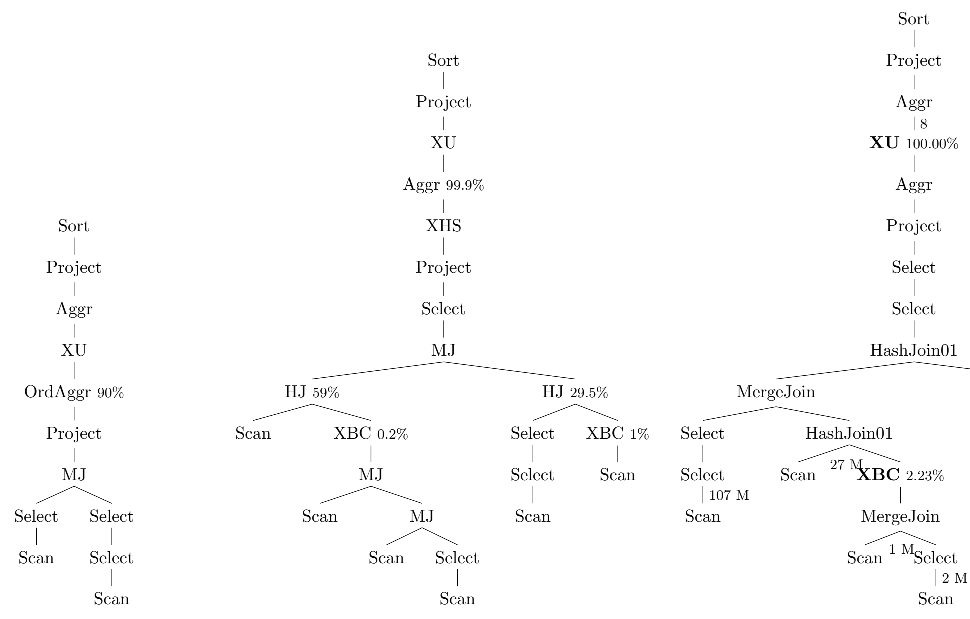
Query four achieves decent speedups. Once again we exploit a join-range index, which enables us to split ranges in a content-aware manner producing clustered and uniquely partitioned ranges. The Xchg operator is put above OrderedAggregation, which also requires clustered output. This was only possible because the join keys of MergeJoin are compatible with the *groupby* list of attributes.

Query 5

In this query, the XchgUnion operator was also put relatively high in the QET. As a result, about 99 % (subtrees below XchgBroadcast operator) of the execution cost was parallelized. This query would benefit much more if we were able to reduce the total computation cost, mainly by reusing once calculated hash tables (calculations performed by HashJoin operators sum up to about 90% of the query execution time).

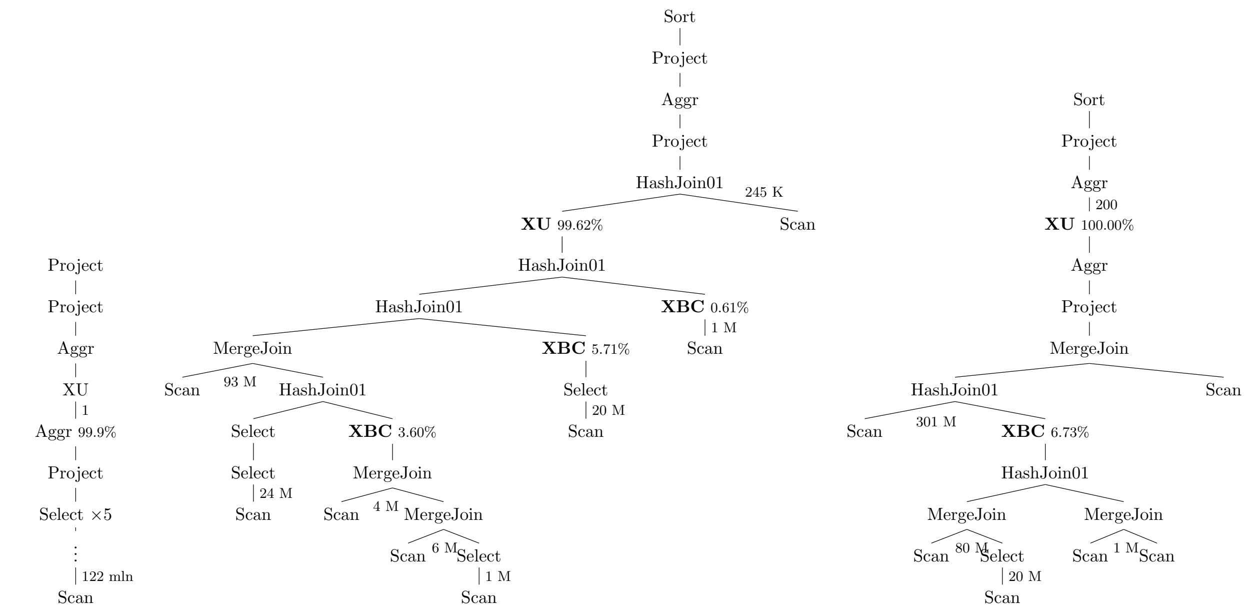
Query 6

The sixth query has a QET very similar to query one. The way of making this tree parallel is the same (using XchgUnion with an additional Aggregation operator). However, the speedups are not as impressive, since the processing time is much lower (17.7 seconds vs. 0.83 second). For the higher number of threads, the fixed cost of creating threads and buffers becomes significant.



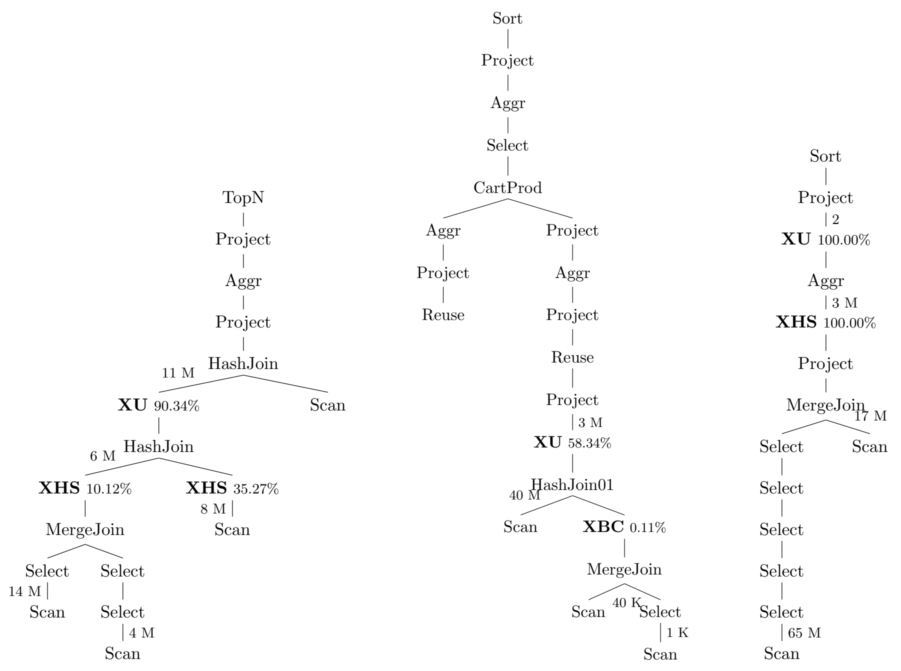
(a) Query 4 (b) Query 5 (c) Query 7

Figure 7.1.3: Parallel QET used in TPC-H benchmark.



(a) Query 6 (b) Query 8 (c) Query 9

Figure 7.1.4: Parallel QET used in TPC-H benchmark.



(a) Query 10 (b) Query 11 (c) Query 12

Figure 7.1.5: Parallel QET used in TPC-H benchmark.

Query 7

In the seventh query we are able to use join-range index to partition the data for about 97.5 (100 - 2.5) % of the computation.

Just like in the previous scenarios, we can limit the total processing cost by sharing the hash tables between operators.

Query 8

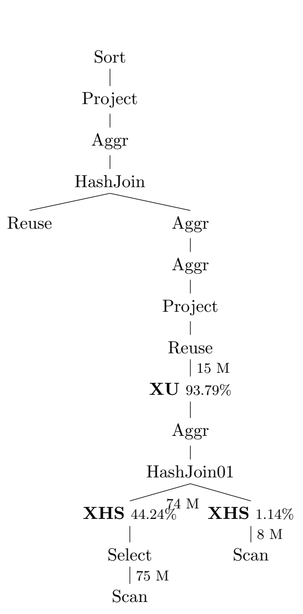
In query eight we obtain a decent speedup after using 2 new threads. Because of the building and synchronization costs the optimizer decides not to use more threads, whereas using four threads is the optimal decision.

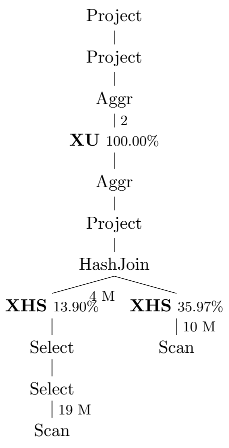
Query 9

About 93 % of the computation cost gets parallelized. Join-range index used to divide ranges for MergeJoin. When 8 cores are used, the processing cost of the the subtree having XchgBroadcast operator as a root node increases to more than 20 %. This is also one of the most expensive queries. We decreased the processing time from about 48 to 9 seconds.

Query 10

Query ten is the first in which we use the XchgHashSplit operators. In this QET optimizer focuses on the most expensive operator - HashJoin with a large inner input. This operator consumes in the


(a) Query 13


(b) Query 14

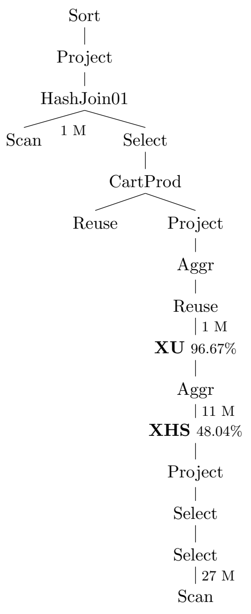

(c) Query 15

Figure 7.1.6: Parallel QET used in TPC-H benchmark.

sequential plan more than 82 % of the computation time. The XchgHashSplit operator decreases the hash table size (hence the building and also the probing phase). After parallelization with two threads, this operator used almost half ($12 * 10^9$ vs. $21.8 * 10^9$) of the initial number of CPU cycles.

For this QET it seems reasonable to move the XchgUnion operator above its current parent (HashJoin). However, as it was already mentioned in section 6.5, it is now always easy to accurately predict the number of input and output tuples for every operator. In this case, we wrongly predict the number of output tuples from the top-most HashJoin operator, making an error of over 10 millions tuples.

Query 11

In Query 11, because of the existence of Reuse operator, we are able to parallelize only about 58 % of the total cost.

Query 12

Query 12 obtains good results. In the parallel QET we first exclusively parallelize the Aggregation operator (with the XchgHashSplit operators). Second, the input for the MergeJoin operator is split using a join-range index.

Query 13

Query 13 is, beside Query 9, one of the most expensive queries. Its sequential version takes about 37 seconds to compute. In the sequential plan, the lowest HashJoin operator takes 46 % of the whole query. The Aggregation operator above takes 31 %, whereas the Select operator performs expensive

LIKE comparison on strings. The whole subtree takes about 93% of the whole computation time.

Since the *groupby* list of attributes is also the list of join keys for the HashJoin operator, by using the XchgHashSplit on those keys we are able to partition the data for both these expensive operators. Hence, we reduce the number of CPU cycles from 31.8 to 17.6 billions for the Aggregation and from 47.6 to 25 billions for the HashJoin operator.

Reducing the data size each of those memory intensive operators processes increases the cache utilization and reduces TLB misses. As a result, we are able to obtain super-linear speedup for two cores.

The parallelization level ceases to increase at the level of 5.

Query 14

For this query the parallelization level ceases to increase at the level of 2. The results for 2, 4, 6 and 8 cores should be the same.

Query 14 has the same parallelization model as in the previous case. Reducing the data size also results in super-linear speedup.

Query 15

In this query, optimizer uses always the maximum parallelization level allowed. However, the we do not observe any speedups for more than 2 queries. This is caused by small computation time of this query (0.4 sec after parallelization), which makes the fixed costs a significant part in the whole computation (for 8 threads the building time takes about 17 % of the total execution time).

Query 16

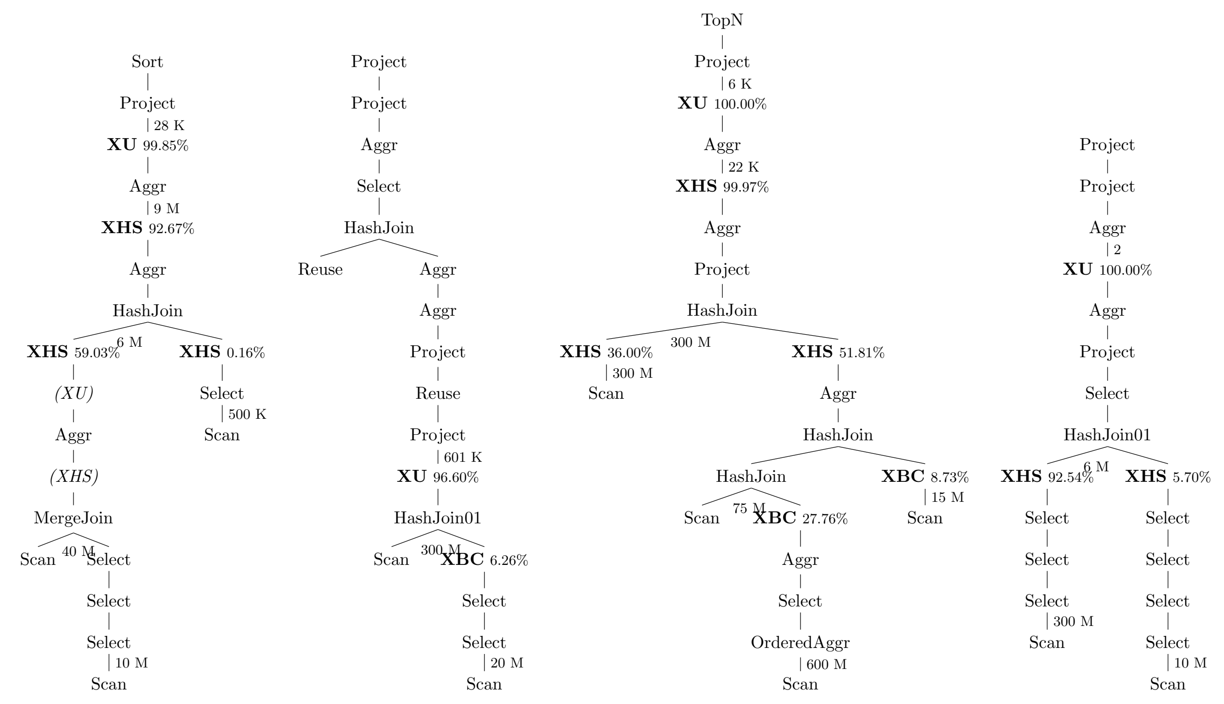
For Query 16, speedups deteriorated significantly when using more threads. The optimizer decided to change the parallel QET in order to decrease fixed building costs (XchgUnion and XchgHashSplit operators in italics). Ceasing to increase the parallelization level with more than 4 threads would be a better decision. Further improvements for cost estimates is an important area for future work.

Query 17

In spite of parallelizing a small number of operators (about one fourth), we managed to partition the data for more than 90 (96.60 - 6.25) % of the computation. In the parallel plan, we divide outer input for HashJoin operator, which initially consisted of 600 million tuples.

Query 18

In Query 18 we are faced with a complicated multi-join execution tree. The optimizer did not parallelize the Aggregation-Select-OrderedAggregation-Scan branch as well as scanning of 15 millions of data. Those two subtrees sum up to about 36% of the whole computation. Also, processing of 300 millions rows (600 millions in parallel) by the XchgHashSplit operator introduces a significant additional cost of $11 * 10^9$ CPU cycles or 22% of the whole computation time.



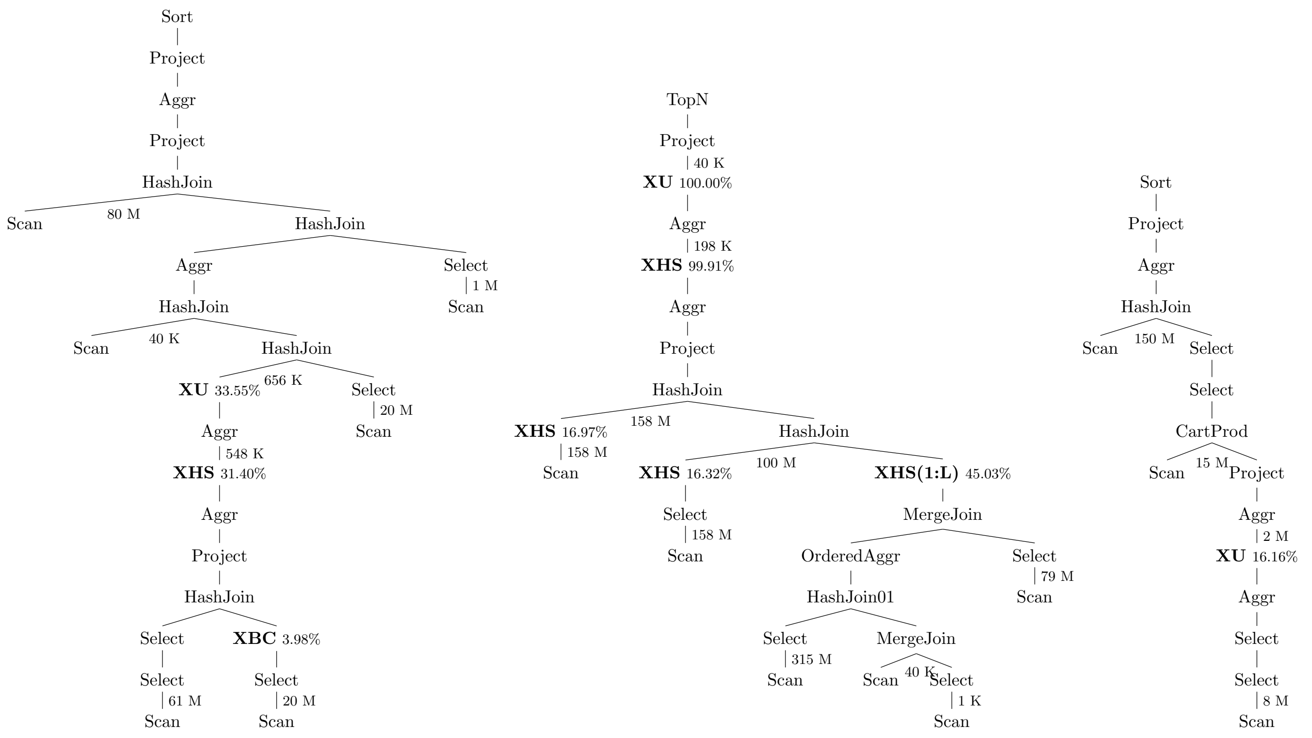
(a) Query 16 - alternative plan used with 6 and 8 threads in italics.

(b) Query 17

(c) Query 18

(d) Query 19

Figure 7.1.7: Parallel QET used in TPC-H benchmark.



(a) Query 20

(b) Query 21 - XHS(1:L) operator uses only one producer and $L \in \{2, 4, 5\}$ consumers.

(c) Query 22

Figure 7.1.8: Parallel QET used in TPC-H benchmark.

Query 19

Query 19 obtains decent speedups, which deteriorate with more threads used. This is mainly because very high operator building costs, which for the parallelization level of eight takes 23% of the whole query.

Query 20

In this query XchgUnion operator cannot be placed above the top-most HashJoin operator, with splitting the outer input and broadcasting the inner input, as for this specific join operator, which returns all the tuples from the right for which there is a matching key on the left, such a transformation is invalid. On the contrary, applying hash split for both inner and outer input (delegating partitioning to children of the next HashJoin operator is in this case invalid as there is no functional dependency between keys used in both join operators) involves the same overhead as in the previous query (processing 600 millions rows in parallel by XchgHashSplit operator). As a result, parallelization was performed very low in the QET so we were able to parallelize no more than 34% of the whole computation time.

Query 21

The subtree rooted in MergeJoin operator required more that 45% of processing time to finish. It was executed sequentially, as it required an (currently not supported) transformation with the

OrderedAggregation below the MergeJoin operator.

Query 22

In this query, the top-most HashJoin operator does not allow to broadcast its inner input (similarly to query 20). We delegate parallelization to a subtree which is not computationally expensive (its processing time is 25 % (16 %) of the query execution time for the sequential (resp. parallel) plan).

Chapter 8

Optimizations

8.1. Thread affinity

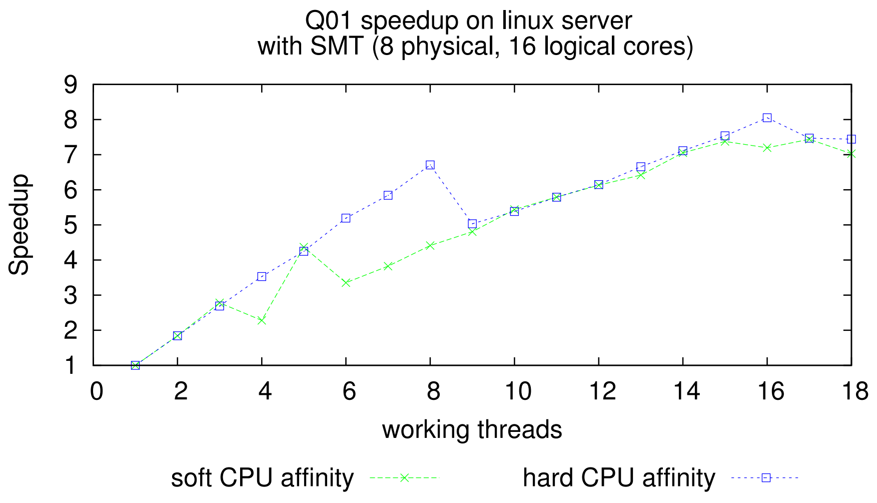


Figure 8.1.1: Q01 with and without hard CPU affinity.

Processes that do not migrate between CPUs frequently, are known to incur less overhead. This is mainly because, parts of the cache data may still not be invalidated, and can be reused after a process regains the CPU. CPU affinity is the tendency of a process or a thread to run on a given CPU as long as possible without being moved to some other processor. The majority of modern operating systems provide algorithms to reduce the frequency of such migration. This is commonly referred to as *soft* CPU affinity.

Both Linux and Windows contain a mechanism that allows developers to enforce *hard* CPU affinity. This means that an application can explicitly specify which processor (or set of processors) a given process may run on.

Hard CPU affinity is especially important when used with SMT. Figure 8.1.1 presents a comparison of the TPC-H Query 1 run with and without (hard) thread affinity. This test was performed on a Linux server machine. We can easily notice that the process scheduler had significant problems with appropriate CPU allocation. This resulted in imperfect speedups, mainly between three to

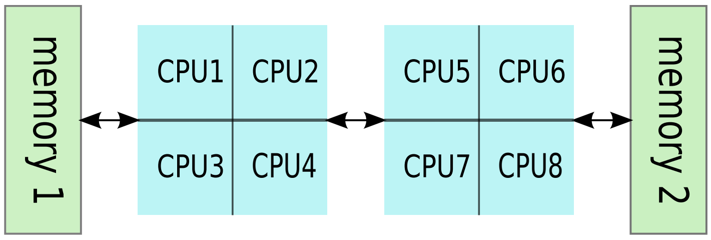


Figure 8.2.1: The NUMA system used in tests.

eight threads working in parallel. Hard thread affinity allocates CPUs more wisely, as we know the relation between logical and physical cores, and the cache usage of a particular query. In this benchmark, We can also notice that once we create nine or more threads, the performance drops and slowly recovers with new threads added. The speedup gains its peak for sixteen threads (with the speedup of 8.05, comparing to 6.7 achieved for 8 threads), proving that SMT is generally worth being used. What is more, if nine or more threads are under operation there is no difference between hard and soft thread affinity. Hard and soft approaches lead to the same workload skew. This is because the work was divided statically between all the threads. Threads sharing the same CPU operate slower than threads having exclusive access to a CPU.

8.2. NUMA architecture

One of the reasons for imperfect speedups, especially for the TPC-H Query 1, is the memory characteristic of the architecture used for tests, which is a Non-Uniform Memory Access (NUMA) architecture. The two physical chips (each having four CPUs) have their own memory control units (fig. 8.2.1) If a thread running on chip A accesses memory that is controlled by chip B it accesses a non-local memory.

On NUMA systems, better performance is obtained by executing processes as close to the memory they access as possible. Each access a process makes to remote memory reduces the performance of that process. Accessing remote memory may also reduce the performance of other tasks, by causing contention for remote memory connections.

NUMA awareness within the scheduler is necessary in order to support the principle of the locality of memory access.

The scheduler of the Linux kernel supports NUMA-aware process allocation. First, once a process is running on a given processor. Linux prefers to keep it on the same node. This rule is combined with the policy of allocating memory from the local node, this helps keep the ratio of local memory accesses high.

In the implemented version of parallelism the majority of memory is allocated in a building phase, which finishes before Xchange operators create their threads. As a result, if a new thread runs on different may chip it accesses non-local memory which penalizes the performance. Conclusions from this observation are discussed in chapter 9.

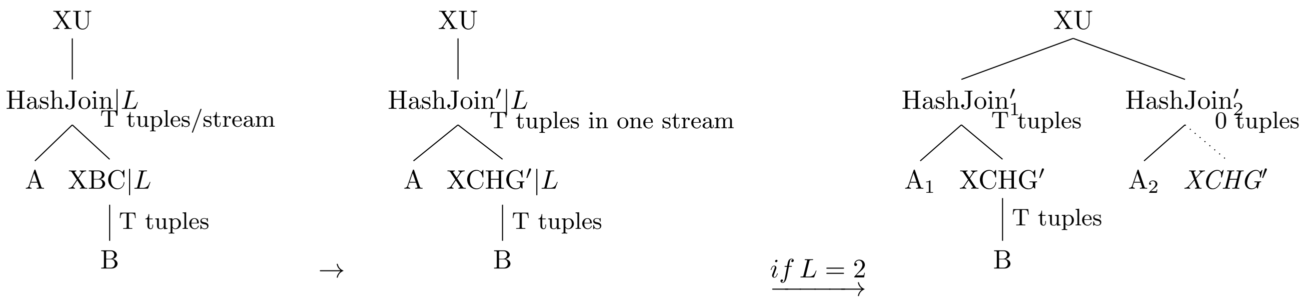
8.3. HashJoin and XchgBroadcast

Transformations using the HashJoin and the XchgBroadcast operators construct the same hash table for each of the join operators. This suboptimality increases the total execution cost and affects the response time (because of the resource contention).

We can deal with this problem in several different ways, which vary in the effectiveness and in the simplicity of the implementation. Below we present three different solutions.

8.3.1. Read-only hash table

In the simplest solution we create the hash table only once and redistribute it (before the probe phase starts) among all HashJoin operators. The below transformation is a replacement for HashJoin + XchgBroadcast transformations. It requires modifications of both operators.



The HashJoin operators form a group of operators which share the state. The first of the operators computes the hash table, whereas the rest of the operators waits on a monitor variable. After the hash table is done all the other operators from the group are awoken and proceed directly to the probe phase.

The XchgBroadcast operator is replaced by the Xchg operator, because only to the first consumer processes the data (the XchgBroadcast operator requires all consumers to process the data, otherwise buffers get clogged).

There are two major drawbacks of this solution. First, it involves changes in the HashJoin operator (this does not conform to the principles of the Volcano parallelization model). Second, in spite of the fact that the total execution cost is decreased, the hash table is still build sequentially by one thread only and the other threads are blocked.

8.3.2. Using the Xchg operator

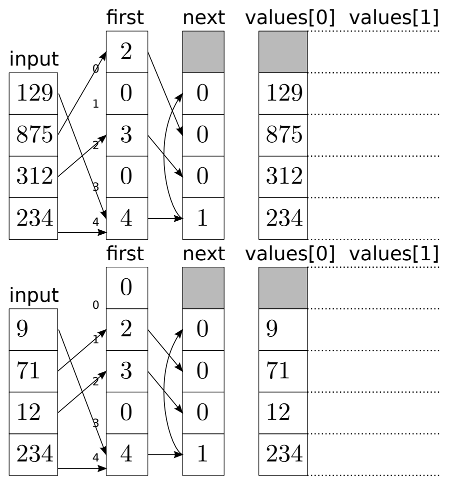
The second solution aims at parallel hash table build phase. It requires more changes in the implementation of the HashJoin operator than the previous solution.

In the VectorWise DBMS the build phase uses a simple bucket-chained hash table (fig. 8.3.1a). The *next* array represents the linked list of all tuples that hash into a given bucket. The zero value represents the end of the list.

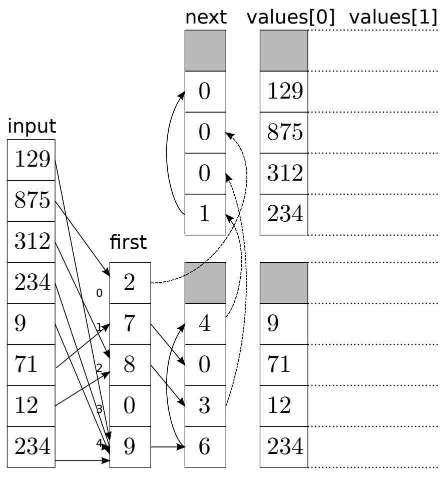
In the figure 8.3.1a two HashJoin operators created two separate hash tables. The inner input was distributed by the Xchg operator. After the build phases of the HashJoin operators (each HashJoin operator builds its hash table in parallel) finish we execute additional computation which joins separate hash tables into one hash table. This is done in the linear cost and is performed by algorithm 6.

The resulting hash table is a table of a lower quality, as the average size of a chain is longer by a factor of the number of initial hash tables. This creates an additional overhead in the probe phase.

We can avoid this problem by creating larger hash tables in the parallel build phase. As a result, the build phase is slower (as we access larger memory), but we obtain possibly better resulting hash table.



(a) Hash tables created by two HashJoin operators working in parallel.



(b) A single bucket-chained hash table - a join of two separate hash tables.

Figure 8.3.1: A simple bucket-chained hash tables. Using *modulo 5* as a hash function.

Algorithm 6 Joining the chains of N separate bucket-chained hash tables.
First, we amend the offsets so the old, local, values are true in the larger hash table. Second, if we encounter a zero value (an indicator of an end of the chain) we try to join the current chain with the corresponding chains.

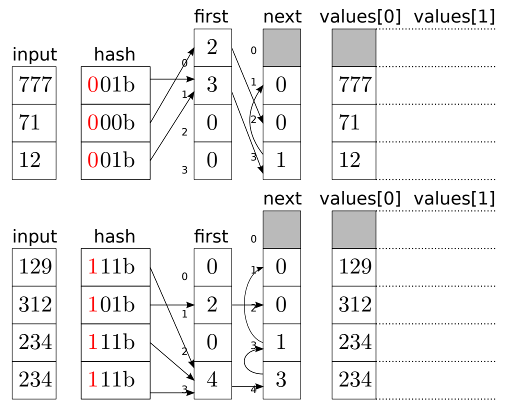
```
1: $offset \leftarrow 0$
2: for $i \leftarrow 2$ TO $N$ do
3:     $next\_size \leftarrow$ sizeof($next_i$)
4:     for $j \leftarrow 1$ TO $next\_size$ do
5:         // $h$ is where the previous list starts (if it starts)
6:         $h \leftarrow hash(input[\ j\ ])$
7:         // if $j$ is the end of some list, try to link it to the previous list
8:         if $next_i[\ j\ ] == 0$ AND $next_{i-1}[\ h\ ] \neq 0$ then
9:             $next_i[\ j\ ] \leftarrow next_{i-1}[\ h\ ] + offset$
10:    $offset \leftarrow offset + next\_size$
```

8.3.3. Using the XchgHashSplit operator

Different HashJoin operators can also produce disjoint hash tables, for which we do not have to correct the hash table's chains, as in the previous solution. This can be done using the XchgHashSplit operator in the place of the XchgBroadcast operator. The XchgHashSplit operator computes the same hash function as calculated in the HashJoin operator and distributes the tuples depending on the *(i)* lower or the *(ii)* higher bits of the tuple's hash value.

Let's assume that the number of streams L is the power of two i.e. $L = 2^j$. The HashJoin operators create hash tables of a size of 2^i (the i value is chosen in such a way that the hash table is twice of the size of the estimated input*). If the XchgHashSplit operator divides the input depending on the higher bits then the addressee of a tuple is calculated from the $i, \ldots, i+j-1$ bits (numbering starts from zero). For example, in figure 8.3.2a the decision about the addressee is defined by the second (red in figure 8.3.2a) bit from the tuple's hash value. If the decision is made from the lower

*This provides a good trade-off between memory-consumption and the probability of conflict [CLRS01].


(a) Two hash tables constructed in parallel. The input was split by the XchgHashSplit operator depending on the 2nd (starting from zero) bit of tuple's hash value. The bucket of the key was determined using *modulo 4* function.

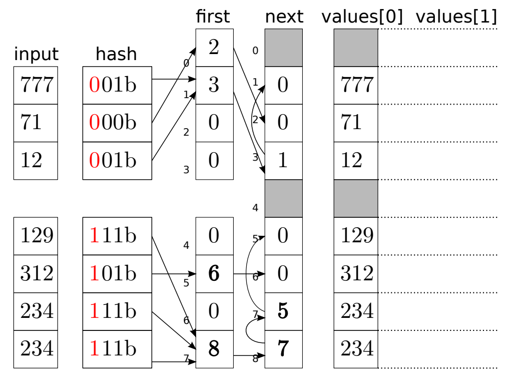

(b) Obtaining the resulting, single hash table is done by concatenation of the partial tables and fixing the offsets (values in bold).

Figure 8.3.2: Using the XchgHashSplit instead of the XchgBroadcast operator.

bits, the addressee is obtained from $0, \ldots, i - 1$ bits.

In this solution we highly rely on the estimated cardinality of the inner input size. This is because the sizes of the partial tables (those that will be concatenated later on) have to be equal. If the partial hash tables were of different sizes, the table concatenation becomes a complicated process, as the hash value calculated for the partial table is unrelated to the hash in the resulting hash table. This means that if any of the HashJoin operators has to rebuild its hash table all hash tables have to be recreated from a scratch and using all the values. The requirement about maintaining the same size of the hash tables becomes a problem in scenarios with bad cardinality estimates or a significant data skew.

8.4. TPC-H optimizations - summary

In this section we summarize the possible improvements for the TPC-H queries. The main areas of improvements are: *(i)* cost estimates (of operators building phase, processing cost and resource contention), *(ii)* implementing parallel build phase, *(iii)* improving the HJ-XBC transformation inefficiency, *(iv)* implementing new transformation rules for improving parallel QET. For each query we identify the most important optimizations.

Query 1 Because the SMT was actually able to achieve better speedups (16 logical cores obtained the speedup of 8.05, when compared to the speedup of 6.7 for 8 physical cores), which is shown in figure 8.1.1, we deduct that this query suffered from stalled processors (due to a cache miss, branch misprediction, or data dependency). Thanks to the SMT another scheduled thread was able to use processor's idle resources. NUMA-aware memory allocation may partially address those problems.

Query 2 Parallel building phase may significantly reduce the query cost. The building phase currently requires up to 30% (for 6 threads) of the query processing time to finish.

Query 3 Optimizing the HJ-XBC transformation may reduce the total execution cost for this query. The parallel plan suffers from recreating the hash table of more than 3 million rows. The maximum parallelization level of 2 is too low. The level of 4 decreases the execution time to 0.47 of a second. With more threads, there is no improvement noticeable.

Query 4 A new transformation rule, that considers functional dependencies between two lists of aggregates may help to make the top-most Aggregation operator parallel (the cost of this operator is about one third of the query execution time for higher level of parallelism).

Query 5 The building phase for the parallelism level of 8 is relatively costly and takes about 15% of the query execution time. What is more, we duplicate hash tables of more than 3.5 million tuples as a result of the HJ-XBC transformation.

Query 6 The building phase for the parallelism level of 8 is takes about 10%, where more than 90% is spent in building of the XchgUnion operator.

Query 7 The HashJoin operators duplicate the work of creating hash tables of more than 1.2 million tuples as a result of using the XchgBroadcast operator. Faster building phase may help to prevent the rewriter from moving the parallelization down in the parallel QET (which results in a less optimal plan, but with a smaller fixed cost). The level of parallelism for this query should cease at 4 (it stops at 5), as the performance deteriorates slightly for the higher number of threads.

Query 8 The maximum parallelism level for this query (as decided by the rewriter) is 2. Operator and resource contention cost estimates should support the decision of using 4 more threads. The parallelism level of 6 yields the performance of 1.2s (about 2 times better than with 2 threads). The performance deteriorates slightly for 8 threads (1.5 seconds). This query recreates hash tables for about 4 million tuples.

Query 9 The parallel implementation of the HJ-XBC transformation will reduce the cost of recreating hash tables of 4.5 million tuples.

Query 10 This query used the parallelism level of 3 at most, which was a good decision. Incorrectly estimated cardinality of the output of the top-most HashJoin operator results in the decision of not making this operator parallel.

Query 11 The Aggregation operator on the path between the CartProd (Cartesian Product) operator and the Reuse operator takes about 20% of the query execution time. A new transformation rule for parallelizing the CartProd operator and proceeding back to the sequential execution before the Reuse operator may help to speedup this query.

Query 12 The building phase becomes about 19% of the execution time for the parallelism level of 8.

Query 13 The maximum level of parallelism for this query was 5, which was a good decision as using 6 threads does not decrease the execution time and using 7 threads makes the query 2 seconds slower (using 8 threads increases the execution time by 7 seconds). The reason for the deterioration is the increased total execution cost which results in resource contention.

Query 14 The maximum level of parallelism for this query was 2, whereas the optimum is 4 (1.2 seconds vs. 1.7 sec). For the parallelism level of 5 the execution time increases to 1.5 seconds.

Query 15 The building phase for the parallelism level of 8 takes about 20% of the query execution time. The rewriter should use the parallelism level of 2 (as it provides the same speedup as 8).

Query 16 The rewriter should use the parallelism level of 4 not 8. Increasing the parallelism level results in performance deterioration. We need better cost modeling for this purpose.

Query 17 Parallel hash table build in the HJ-XBC transformation can slightly improve the performance (the hash table in the HashJoin operator consists of about 20,000 tuples).

Query 18 The parallel QET duplicates hash tables having more than 15 million tuples (the HJ-XBC transformation). New transformation rule that identifies functional dependencies between two Aggregation operators may help to parallelize costly (about 27% of the execution time) XBC-Aggr-Select-OrderedAggr-Scan subtree (see figure 7.1.7c). Also for the given parallel QET the maximum level of parallelism should not be higher than 4.

Query 19 The building phase becomes 27% of the query execution time for the parallelism level of 8.

Query 20 Because of the problems described in section 7.1 it is not trivial to find optimal parallel QET. The top-most HashJoin operator, called HashRevSemiJoin, returns all the tuples from the right for which there is a matching key on the left. Because of that the standard HJ-XBC transformation does not apply here (two tuples having the same values on the key columns may be placed in two different streams, which results in two tuples being matched instead of one). On the other hand, the solution with the XchgHashSplit operator suffices, but is costly (a lot of tuples have to be copied and split across the HashJoin operators).

A new transformation with additional Aggregation operator for duplicate elimination is a solution to consider.

Query 21 A new transformation analyzing functional dependencies between the MergeJoin operator and the OrderedAggregation operators might make the subtree below the XchgHashSplit(1:N) operator parallel. This subtree executes in 45% of the total query execution time.

For the current plan the parallelism level should cease to increase with more than 4 threads. The rewriter needs better cost estimates and resource contention estimates for that purpose.

Query 22 A new transformation (as described for Query 20) may result in parallelizing larger subtree (more than 99% versus 16% which is already made parallel).

Chapter 9

Conclusions and future work

In this master thesis we presented the implementation of multi-core parallelization of the vectorized execution model. We created a framework for parallelization, which implemented a set of Xchange operators, a set of transformations for query execution trees and a strategy for traversing the state space of parallel query execution trees under a simple cost model. The framework was based on the Volcano model.

We also measured the performance of the implemented solution using the TPC-H benchmark. Tests run on the 180GB of data. Table 9.1 presents the summary of the obtained results.

	1 thread	2 threads	4 threads	6 threads	8 threads
total time	215.7	118.9	87.36	81.19	80.1
speedup		1.81	2.47	2.66	2.69
efficiency		91%	62%	44%	34%

Table 9.1: The summary of the obtained results in the TPC-H benchmark.

The framework proved to be sufficient for providing automatic intra-operator parallelization, although we encountered performance inefficiencies for a bigger number of threads allowed.

Some of the problems can be addressed in the future. The main optimization issues are:

(i) sharing hash table by the HashJoin operators,

(ii) fast implementation of the XchgDynamicSplit and XchgDynamicHashSplit operators,

(iii) extending the set of available transformations,

(iv) decrease the Xchange operators building time,

(v) make the building phase NUMA-aware,

(vi) make the building phase parallel,

(vii) make the number of buffers and their size dynamic,

(viii) improve operator cost estimates,

(ix) improve resource contention modeling,

(x) identify performance bottlenecks and suboptimal implementation.

Some features, like implementing the parallel build phase, the NUMA-aware memory allocation or parallel hash table build are tasks with a well defined goal. Contrary, improving the cost estimates, contention modeling or implementing new transformations cannot be simply accomplished, as we can constantly try to make our models better.

Acknowledgements

I would like to thank my supervisors Peter Boncz and Henri Bal for all the guidelines, inspirations and priceless comments I was given.

This master thesis was made possible thanks to Marcin Żukowski and the VectorWise company where I was offered an internship. It was also Marcin who was always willing to help, who gave good advices, pointed out bad decisions and who participated in all my ideas.

I would like to thank Michał, for his answers to my questions and a great time after work and also to Gosia, Hui, Giel and Sandor. I have learned a lot from all of you and I have been working in a very friendly environment.

I would like to thank my family for their support and the certainty that I can always depend on them.

Last but not least, thanks to Kasia, for understanding, patience, encouragement and simply being.

Bibliography

[BZN05] P. A. Boncz, M. Zukowski, and N. J. Nes. Monetdb/X100: Hyper-Pipelining Query Execution. In *Proceedings of International conference on verly large data bases (VLDB) 2005*. Very Large Data Base Endowment, 2005.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. McGraw-Hill Book Company, Cambridge, London, 2. edition, 2001. 1. editon 1993.

[CRG07] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN '07: Proceedings of the 3rd international workshop on Data management on new hardware*, pages 1–10, New York, NY, USA, 2007. ACM.

[GCD$^+$91] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz. Extensible query optimization and parallel execution in volcano. In *Query Processing for Advanced Database Systems, Dagstuhl*, pages 305–335. Morgan Kaufmann, 1991.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. *SIGMOD Rec.*, 21(2):9–18, 1992.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. *SIGMOD Rec.*, 18(2):358–366, 1989.

[HNZB07] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *In Proc. of the Third International Workshop on Data Management on New Hardware*, 2007.

[HZN$^+$10] S. Héman, M. Zukowski, N. J. Nes, E. Sidirourgos, and P. A. Boncz. Positional Update Handling In Column Stores. In *Proceedings of ACM SIGMOD International Conference on Management of Data 2010*. ACM, June 2010.

[JPA08] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Critical sections: re-emerging scalability concerns for database storage engines. In *DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 35–40, New York, NY, USA, 2008. ACM.

[KKL$^+$09] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[Man02] S. Manegold. *Understanding, Modeling, And Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam (UvA), December 2002.

[TPC] TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[vB89] G. von Bultzingsloewen. Optimizing sql queries for parallel execution. *SIGMOD Rec.*, 18(4):17–22, 1989.

[WFA95] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. *SIGMOD Rec.*, 24(2):115–126, 1995.

[Żu09] Marcin Żukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, CWI, UvA, 2009.