Vrije Universiteit Amsterdam
Departament of Computer Science

Anna Jančaříková
Student number: 1352695

Distributed query processing over a peer-to-peer network

Master's Thesis

Supervisors:

Dr. Peter A. Boncz
CWI

Prof. Dr. Maarten van Steen
Vrije Universiteit

17th July 2003

The research described in this thesis was performed during the author's internship contract at Centrum voor Wiskunde en Informatica - National Research Institute for Mathematics and Computer Science in the Netherlands, within the theme Data Mining and Knowledge Discovery, a subdivision of the research cluster.

This thesis is ready to be marked.

Date: Author's signature:

This thesis is ready to be verified by the second reader.

Date: Supervisor's signature:

Contents

Abstract

In the last years, Peer-to-Peer systems have been in the centre of the interest of the internet community. The goal of AmbientDB is to provide full relational database functionality for autonomous devices connected in ad-hoc network. This thesis focuses on one of many aspects of the system - Query Processing. The main goal is to formulate a query processing framework and implement a network simulation of it.

Chapter 1

Introduction

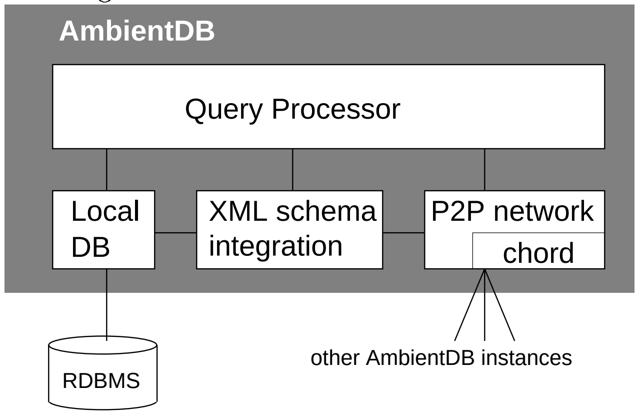
Relational databases have played a central role in information technology during the last 30 years. They typically follow a centralistic architecture, or in the case of distributed databases, a fixed and well-planned topology. During the last years, Peer-to-Peer systems like Napster have been in center of the interest of internet community. The main idea of such systems is to create a decentralized community of machines, pooling their resources. Addition of such systems are: scalability, robustness and lack of need for a central administration. The idea in the new field of P2P databases is to combine the best of these two worlds: powerful query processing functionality and the ad-hocness and scalability of P2P systems.

Information technology continues to expand to diverse sectors of our lives. We can now find computers also in devices where we do not expect them. Almost all electrical appliances have some processor or chip with their own memory and own logic given by a program. A common household is full of such devices like phone, PDA, PC, laptop, digital camera, stereo, microwave, central heating,... . According to the vision of "ambient intelligence", such devices will become more and more connected to each other with new wireless communication technologies such as Bluetooth and 802.11. The applications running on them as well as their user interfaces and applications will become more intelligent and provide automatic cooperation between these connected devices. Such a network could be very heterogeneous with big differences between nodes and with different ways of connecting separate nodes. Its topology should preferably be dynamically self organized. The organization of such a network could be peer-to-peer where no centralized control exists and each node can be a client and a server at the same time.

This thesis explores data management and in particular query processing over such dynamically organized P2P networks. In peer-to-peer systems each peer can share data with other peers.

In peer-to-peer systems, each node can be different and the overlaying distributed database system has to use a strategy which is aware of hardware limitations and provides transparency to hide different hardware and data representations.

Figure 1.1: AmbientDB Architecture



Local DB is a wrapper around any (external) data store.

1.1. AmbientDB

The AmbientDB [4] is a project at CWI performed in association with Philips, where the target is to support ambient intelligent applications for networked devices. The goal for AmbientDB is to provide full relational functionality for autonomous devices. The intention is to enable such devices to cooperate in an ad-hoc way with other AmbientDB devices. Mobile devices have typically few resources (CPU, memory, network) so AmbientDB should take these limitations into account. The AmbientDB architecture shown in Figure 1.1 includes a network peer-to-peer protocol, Distributed query processor, local DB component and schema integration.

1.1.1. Assumptions

We now outline the main assumptions made in the AmbientDB for query processing.

Topology

The topology of the network is a connected graph, where nodes are devices and edges are connections between them. This topology is very general and for communication will be used an overlay tree topology - a spanning tree G(V,E). The node posing a query is (from its own viewpoint) the root of this tree. For each node n there exist an isomorphic graph G'(V,E) where n is the root node. An important assumption for now is that while a query executes, none of the participating devices will disconnect. The AmbientDB project will need to address this issue in the future, as in P2P networks participating devices connect and disconnect in a dynamic fashion.

Global Schema

Each device can have different data representations. However, we suppose a wrapper on each database that masks these differences and provides the view that all nodes have the same database schema. The goal of the system is to answer global queries over the schema where the global table contents are the union of the table content of all nodes on the tree. The global schema integration should work without centralized administration. New connected node first updates own schema if its version is smaller than actual version in the network.

Resources

Resources may be very scarce on some nodes. Small devices have small memory, and our proposed algorithms have to respect this limitation. Also some nodes are assumed to have a high network latency and a scarce network bandwidth. Overall, we assume that network cost will be the dominant factor in query processing. This justifies the use of the ns2 simulator [7, 1] , which only measures network cost.

Replication

Some data tuples will be replicated over the network. Query processing has to be aware of possible existence of duplicates of each item.

Upscaling

The amount of cooperating devices can be potentially large.

1.2. Related work

First, file sharing P2P systems are introduced and then relational database systems are discussed.

1.2.1. File sharing

P2P systems are widespread for file sharing. Well known are Napster, Gnutella and KaZaA and they support searching files that name contains a given string.

Peer-to-peer systems architecture can be categorized in one of the following groups [18]:

Chained Architecture Servers form a linear chain that is used for answering queries. When a user submits a query, the local server attempts to satisfy the query alone. However, if the local server cannot find the maximum number of results, it will forward the query to the remote server along the chain.

Full Replication Architecture Forwarding queries to other servers can be expensive. The full replicated architecture avoids these costs by maintaining a complete index on each server, so that all queries can be answered at a single server.

Hash Architecture Index is hashed to different servers. When a user submits a query, we assume that it is directed to a server that contains at least a part of the result. That server then asks the remaining servers involved for the rest of the result and when the result arrives, it is merged in usual fashion to produce the result.

Unchained Architecture This architecture simply consists of a set of independent servers that do not communicate with each other. A user, who logs in to one server, can only see the files of the users on the same server.

Napster uses a central server to maintain a list of shared files and keeps information about active connected peers. When a file is required then it is downloaded directly from the owner of the file.

Gnutella system is fully decentralized. Each node keeps connections with its neighbors and forms an overlay network. The topology of this network is a spanning graph. Queries are propagated to neighbors by a TTL (time-to-live) protocol, and after n hops are discarded. This controlled query flooding decreases the probability of successful search. Query flooding is the reason why Gnutella has lack of scalability. With the rising number of nodes, the number of queries rises. Then there are more queries per node and the response is as slow as the slowest link.

KaZaA combines the main ideas of Napster and Gnutella. Its overlay network is built from more powerful nodes called "SuperNodes" and other nodes are connected to their nearest SuperNode. SuperNodes maintain lists of files of their neighbors and handle their queries like in Gnutella. Supernodes are selected automatically, but the precise protocol has not been disclosed.

Distributed Hash Tables (DHT)[11] are proposed to improve string searching queries in Peer-to-Peer file-sharing systems. The indexing scheme is to split each string into "n-grams": distinct n-length substring. These items are inserted into the hash index, indexed by n-grams. String required in query is also split into n-grams. Condition for result of containing these n-grams is not sufficient, because these n-grams may not occur consecutively and in the correct order. To achieve final result items have to be directly tested for substrings.

P2P filesharing systems are popular and large scale, but they do not support complex queries.

1.2.2. Relational databases

A vision of future usage of Data management for P2P systems is presented in [2]. This model assumes that data in a P2P network are stored in local relational databases. Network topology is defined by a set of acquaintances. Each acquaintance describes coordination formulas defining dependencies between the two databases.

Piazza system [8, 10] is built at the University of Washington and its vision is peer data management system (PDMS) that provides “semantic mediation” between environment of thousands of peers, each with its own schemes. Rather than requiring the use of a single, uniform, centralized *mediated schema* to share data between peers, Piazza allows peers to define semantic mappings between pairs of peers. In turn, transitive relationships among the schemes of the peers are exploited so the entire resources of the PDMS can be used. In this M.Sc. thesis, schema integration problem is not solved yet.

Sensor networks [5] are networks of very small devices such as temperature sensors, motion detectors lights, or door locks. Typical queries are historical queries (typically aggregate queries) or snapshot queries (query the state in a given time). In [14], a query language, that executes aggregation queries on tree network topology, is proposed. Sensors run Tin's and they are connected to an ad-hoc network. Devices can identify each other and route data without prior knowledge about the network topology. Tiny AGregation (TAG) queries are distributed into the network by piggybacking on the existing ad-hoc networking protocol. Sensors route data back toward the user thought a routing tree. As data flows up this tree, it is aggregated. Our QP framework is inspired with this “in network” paradigm.

Pico DBMS [3] is an example of an implementation of query processing with hardware limitations imposed by smartcards. Smartcards have very slow write operations and very little RAM. The proposed query execution model does not require RAM and Select-Project-Join queries are executed by pipelining.

Traditional distributed database technology works on a known collection of participating sites and communication technology. Peer-to-peer systems do not have this assumption. Federated database technology works on a heterogeneous schema, but schema integration is made by static wrappers and statically configured combinations of databases.

A parallel DBMS aims for improving the query performance by parallel and distributed computation. A dataflow architecture offers the possibility to exploit the inter-operator parallelism and pipelining by allocating different relational operations to different sets of processors. Teradata, GAMMA [6], and PRISMA are examples of parallel DBMSs that were implemented. [17] studies inter-operator parallelism strategies on PRISMA. Assumptions for such systems are different from our work: Parallel machines are well-tuned, well-controlled and well-connected. Dataflow usually works with point-to-point communication between all nodes. This is not sustainable in internet-scale and with the worse quality of internet connections. We inspire us with an dataflow architecture on systems such as PRISMA.

Article [13] gives an overview over new trends in distributed query processing. Distributed systems could be very large. That is why new techniques that presume replication and caching of data and that reduce communication costs are necessary.

1.3. Problem Statement

1.3.1. QP Framework

This thesis is focuses on one of many aspects of Peer-to-Peer systems - Query processing. The main goal is to formulate a query processing framework and implement a network simulation for it. We inspire our QP framework with the dataflow architecture and “in network” processing.

1.3.2. Outline

This Thesis is organized as follows. Chapter 2 summarizes the foundations of Relational Algebra. In Chapter 3 we propose the design of a Distributed algebra and a distributed query processing framework. In Chapter 4 we describe the implementation of our prototype simulation on the network simulator ns2. Experimental results are presented in Chapter 5. Future work and conclusion is in Chapter 6.

Chapter 2

Preliminaries

2.1. Relational database concepts

Definition: A *database* is a structured collection of data. *Relational database* is one with relations as data structures. *Relation R* is defined over n sets $D_1, D_2, ...D_n$ is a set of n-tuples (*tuples*) $\langle d_1, d_2, ..., d_n \rangle$ such that $d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n$. $d_1, d_2, ..., d_n$ are *attributes* and $D_1, D_2, ..., D_n$ are *domains*.

2.2. Relational algebra

Relational algebra consists of five fundamental operators that operate on relations and their product is again relation.

Selection (Unary operator) Selection returns a subset of relation R that satisfies boolean formula F.

$$\sigma_F(R)$$

Projection (Unary operator) Projection returns a vertical subset of relation. The result contains only those attributes which are enumerate in projection parameters. The projection of relation R over S, where S is subset of attributes of relation R:

$$\pi_S(R)$$

It is possible to define projection with or without duplicate elimination, because this definition does not specify it.

Union (Binary operator) The union of two relations R and S is the set of all tuples that are in R, or in S, or in both.

$$R \cap S = \{x : x \in S \vee x \in R\}$$

Set difference (Binary operator) The set difference of two relations R and S is the set of all tuples that are in R but not in S.

$$R \setminus S = \{x : x \in R \land x \notin S\}$$

Cartesian product (Binary operator) The Cartesian product of two relation R and S is the set of tuples that are created by concatenation of one tuple from R and one tuple from S, for all tuples of R and S.

$$R \times S$$

Additional operations are derived from fundamental operations.

Join Join is derived from Cartesian product and selection. The join of two relations R and formula F is denoted as

$$R \bowtie_F S = \sigma_F(R \times S)$$

Natural join Natural join is equi-join of two relations over a set of attributes R and S with the same domain.

$$R_A \bowtie_B S$$

Aggregate Functions Aggregate operators are extension of projection.

$$\pi_{S,f(A)}(R)$$

Where S is subset of attributes of relation R and f is function operation on domain of attribute A.

Properties:

Commutativity

$$R \times S = S \times R$$

$$R \bowtie S = S \bowtie R$$

Associativity

$$(R \times S) \times T = R \times (S \times T)$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Idempotence

$$\sigma_{F_1}(\sigma_{F_2}(R)) = \sigma_{F_1 \land F_2}(R)$$

$$\pi_{A_1}(\pi_{A_2}(R)) = \pi_{A_1 \cap A_2}(R)$$

Distributivity

- Projection (Selection) and join

$$\pi_C(R \bowtie_{p(A_i,B_j)} S) = \pi_{C'}(\pi_{A'}(R) \bowtie_{p(A_i,B_j)} \pi_{B'}(S))$$

$$\sigma_{p(C)}(R \bowtie_{q(A_i,B_j)} S) = \sigma_{C'}(\sigma_{A'}(R) \bowtie_{p(A_i,B_j)} \sigma_{B'}(S))$$

  where
$$A' = C \cap A \setminus A_i, B' = C \cap B \setminus B_j, C' = C \cap (A_i \cup B_j)$$
  and A and B are sets of attributes over which are relations R and S defined.

- Projection (Selection) and union

$$\pi_C(R \cup S) = \pi_C(R) \cup \pi_C(S)$$

$$\sigma_{p(C)}(R \cup S) = \sigma_{p(C)}(R) \cup \sigma_{p(C)}(S)$$

Chapter 3

Design

AmbientDB provides global query processing functionality over a peer-to-peer network with its global abstract algebra as query interface. Let us assume a P2P network with participating nodes N_i and table instance T_i for each table T in the global schema. The global abstract algebra works on a global schema of abstract tables, each of which can take one of the following forms:

Local table (LT) is table T_i in isolated view on one node T_i

Distributed table (DT) T_Q is defined over a set of nodes Q as $T_Q = union(T_i), \forall i \in Q$

Partitioned table (PT) is a specialization of the distributed table, where all participating tuples in each T_i are disjunct between all node.

Queries follow a three-level translation (Figure 4.2). User queries are given in Abstract global algebra (Tab. 3.1). Operators of this abstract global algebra are translated to concrete global algebra operators. Concrete operators have a fixed processing strategy, and are processed on one or more bidirectional waves that consist of dataflow algebra operators, which form the lowest level of our query processing framework.

3.1. Abstract Global Algebra

Select Returns new table that contains only those tuples from the input table that evaluates condition to be true. The new table has the same attributes as the input table.

Project Returns a new table consisting of columns corresponding to all projections.

Order Returns a new table where tuples occure in order given by evaluation of expression orderby.

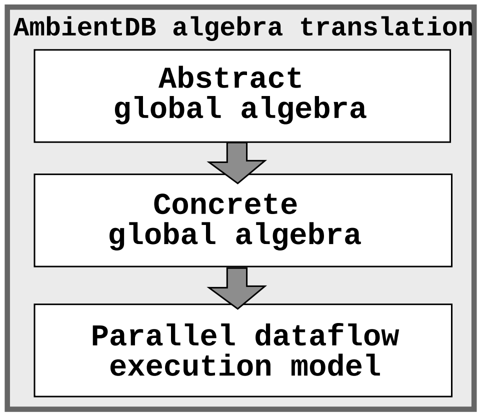

AmbientDB algebra translation
Abstract global algebra
Concrete global algebra
Parallel dataflow execution model

Figure 3.1: 3-level Algebra Translation

Aggregation Aggregation returns a new table consisting of both the computed aggregates and the groupby projections, with one tuple for each combination of groupby expressions in the source table.

fnc = SUM, COUNT, AVG, MIN, MAX ...

Join returns a new table consisting of columns of both tables left and right except those columns that are in expression key proved to be equal.

abstract global algebra
`Select(Table t; Expr cond)→Table`
`Aggr(Table t; List<Expr> groupby)→Table`
`Join(Table left,right;Expr cond)→Table`
`Order(Table t; List<Expr> orderby)→Table`
`Project(Table t; List<Expr> resultColumns)→Table`

data model
`Column(String name; int type)`
`Key(bool unique; List<Column> columns; Table table)`
`Table(String nme;List<Column> cols;List<Key> prim,forgn)`

expressions
`Expr(int type)`
`Expr::ConstExpr(String printedValue)`
`Expr::ColumnExpr(String columnName)`
`Expr::OperatorExpr(String opName, List<Expr>)`

Table 3.1: The Abstract Global Algebra

The artificial column NODEID is added to each table. It holds the node identifier of the node where the tuple is stored (or one of the nodes if the tuple is replicated).

3.2. Concrete Global Algebra

The concrete global algebra (Tab. 3.2) contains corresponding operations to the abstract global algebra plus two extra operations partition and union. Operations of the concrete global algebra correspond with a particular execution strategy.

concrete global algebra **(T_1, $T_2 \in$ {DT, PT})**		
`Union(T1 t; List<Expr> key, result)→LT  #` *`create LT from DT/PT by merging`*		
`Partition(DT t; List<Expr> key)→PT  #` *`create PT from DT by finding duplicates`*		
select$_{local}$(LT)$\to$LT	join$_{local}$(LT,LT)$\to$LT	aggr$_{local}$(LT)$\to$LT
select$_{dist}$(T_1)$\to T_1$		aggr$_{merge}$(T_1)$\to$LT
	join$_{broadcast}$(LT,T_1)$\to T_1$	
	join$_{split}$(LT_1,T_1)$\to T_1$	aggr$_{dist}$(T_1)$\to$DT
order$_{local}$(LT)$\to$LT		union$_{merge}$(T_1)$\to$LT
	join$_{foreignkey}$(T_1,DT)$\to T_1$	
order$_{dist}$(T_1)$\to T_1$	partition(DT)$\to$PT	union$_{elim}$(T_1)$\to$LT

Table 3.2: The Concrete Global Algebra

There are local executions variants (*local*) executed only at query node and distributed execution variants (*dist*), where the operation is broadcast on the network and executed on all nodes. Each node executes the operation on its partition of a partitioned table or distributed table. This process then produces again a PT or DT. Local and distributed variants exists for aggr, select and order.

Join has more variants:

local

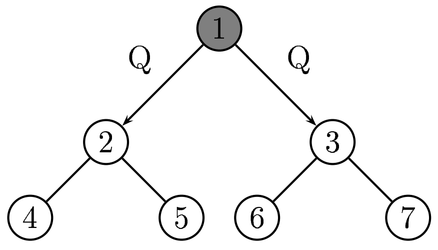
broadcast One of the join tables is broadcast to all nodes and joined in each node with its local table partition.

foreignkey For equi-joins on a foreign key, we can deduce from the referential integrity in each node, that all matching tuples can be found locally. Thus, this strategy only broadcast the query, and each node performs a local join like the dist strategy.

split In this variant of broadcast join, where the join predicate contains an equality condition on NODEID, tuples are routed only to the nodes indicated in the NODEID of the "broadcast" table.

All join, union and aggr suppose their inputs to be ordered.

The union has two variants in concrete global algebra: $union_{merge}$ and $union_{elim}$. $union_{merge}$ is processed "in network". First, the query is broadcast, then each node merges its own local table with the input-stream provided by its children. $union_{elim}$ is also processed "in network" and it eliminates duplicated tuples.



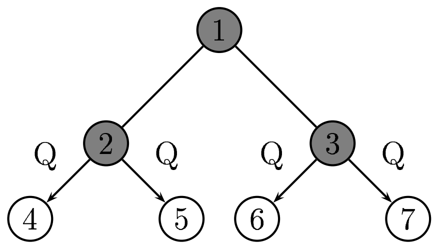


Figure 3.2: Query processing in two waves: the first wave
Query propagation (down - in direction from the root node to the leaves)

$Aggr_{local}$ computes aggregation locally on LT. $Aggr_{dist}$ produces distributed table with aggregates only over a local partition. $Aggr_{merge}$ computes aggregates during propagating in network.

Other concrete algebra operations are

partition The operation partition() creates a partitioned table from the distributed table $T = \cup T_i$.

3.3. Dataflow Algebra

3.3.1. Query processing on a tree topology

The main idea of our query execution algorithms, described later, is processing in waves. A wave is a data stream that is propagated from one node to another node in the direction of the wave.

down A wave from the root node to the leaves is propagated from the node which received this wave to all its child nodes. (broadcast)

up A wave from the leaves to the root is propagated from the node that received waves from each of its children to the parent node. (merge)

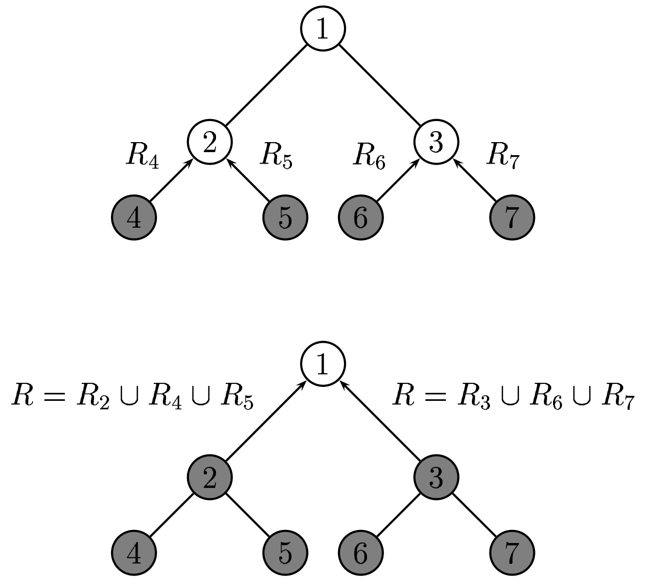


Figure 3.3: Query processing in two waves: the second wave
Result merge (Up - in direction from the leaves to the root node)

Figure 3.2 demonstrates a downward wave propagating a query. Each node that receives this query, forwards it to its child nodes and processes it on its local database. An example strategy that behaves like this is $select_{dist}$. Figure 3.3 gives the upward wave, gathering the results. An example strategy that behaves like that is $union_{merge}$.

3.3.2. Pipelining operations

As it was presented in the section Assumptions, we want to propose a framework that works on a tree topology and does not require uncontrollable memory resources. The solution chosen here is data processing by pipelining. Pipelining is sequential processing of one or more input data streams to one or more output data streams. Data is read sequentially and each item is read exactly once.

All operations presume that input streams are ordered by a hyperset of key attributes.

Processing by pipelining has two additional advantages:

- It reduces I/O cost, because intermediate results need not be written back to the disks.
- The first tuples of the resulting operations can be produced earlier.

On ordered streams of tuples we can define pipeline operators. The output of the pipeline operator is the product of the corresponding set operation on the input streams.

3.3.3. Dataflow Algebra Operations

The dataflow algebra (Table 3.3) contains operators that work with data streams in a pipelined fashion.

	Dataflow Algebra Operators
n	**scan**(Buffer b)→Dataflow
s	**select**(Dataflow d; Expr cond; List<Expr> result)→Dataflow
a	**aggr**(Dataflow d; List<Expr> groupBy,aggr)→Dataflow
o	**order**(Dataflow d; List<Expr> orderby,result)→Buffer
n	**topn**(Dataflow d; List<Expr> orderby,result, int n)→Buffer
j	**join**(Dataflow d$_l$, d$_r$; List<Expr> key$_l$,key$_r$,result) # merge-join on dataflows ordered on *key*
m	**merge**(Dataflow d$_l$,d$_r$; List<Expr> key)→Dataflow # merges *key*-ordered dataflows, returning tuples in order # adds $t.\#cnt$: number of consecutive tuples with equal key # and $t.\#nr$: which ascends 0,1, etc.. in each such chunk
t	**split**(Dataflow d; List<Buffer>$<b_1..b_n>$; List<Expr>$<f_1..f_n>$)→Dataflow # returns equal stream, inserts $t \forall i : f_i(t) = true$ in b_i

Table 3.3: The Dataflow Algebra

scan creates a datasteam from a buffer or LT

select scan the input datastream, outputting only those items from datastream that fulfill the condition.

aggr makes an output stream with aggregated items sorted from the input stream.

order Order makes a buffer from an input datastream. Items in this buffer are ordered by expressions in the orderby-list.

join This operation joins items from inputs streams d_r and d_l that fulfill the condition.

merge Merge makes from multiple key-ordered input datastreams one ordered output datastream and adds two attributes to each item: cnt and nr. Cnt holds the number of tuples with equal key value and nr is rank of items within each group.

split This operation makes n buffers, and fills each with items from the input datastream that fulfill the corresponding condition from the list of expressions.

3.4. 3-level Algebra Translation

The abstract global algebra is a standard relational algebra. A query is transformed to a low-level wave plan which implements a particular execution strategy. A query may have many equivalent and correct transformations and each can lead to different consumptions of computer and network resources.

3.4.1. Translation from Abstract Algebra to Concrete Algebra

Any table in the leaves of an abstract global query graph is resolved to either a LT, DT or PT. According to this, abstract operators are instantiated to concrete operators. Thus we get concrete algebra signatures for the operators in the leaves and we also know the type of the result. The end result have to be a LT. This bottom-up conversion of abstract operators in a query plan to concrete operators is the main mechanism to translate a global abstract query into a global concrete query.

Not all concrete signatures are implemented (i.e. $join(DT, DT) \rightarrow DT$). In this case we can continue by recursion with translation. We have $join_{bcast}(LT, DT \rightarrow DT)$ and we can use the union for translation DT to LT.

One abstract global query has more possible translations in the concrete global algebra. The task of query rewriter is also to chose the optimal query with the lowest execution cost. The execution cost is expressed as a weighted combination of I/O, CPU, and communication costs. We assume that the communication cost is dominant and we can simplify it and ignore local processing cost(I/O and CPU costs). Inputs for estimating execution costs are statistics and formulas for estimating the cardinalities of results of operations.

Some concrete algebra operations require ordered input. So, the rewriter should insert order operations automatically. Such operations are: $union_{elim}$, $union_{merge}$, $aggr_{merge}$ and $join_{bcast}(join_{split})$.

Union

The Concrete Algebra operation Union does not have an equivalent in abstract global algebra. The union is introduced as an auxiliary operation. We can eliminate a DT or PT from the result by using operation union.

$union_{elim}(DT\ T)$ is equivalent to $union_{merge}(PT\ T)$.

When a query requires ordering by $(c_1, c_2, ..., c_n)$ then it replaces this ordering by $(c_1, c_2, ..., c_n, primarykey)$.

The partition operation can be used in the case, we have a DT but we want to use $union_{merge}$ instead of $union_{elim}$. This can be beneficial in the case that the size of the tuple is much bigger than the size of the key attributes and there are a lot of duplicated tuples. Then using partition() can reduce communication. During processing the operation partition only key attributes are sent.

Select

Select has two variants local and distributed. Which variant will be chosen depends on the abstract table in query.

- select(DT)→DT

$$select_{dist}(DT\ T)$$

- select(LT)→LT

$$select_{local}(LT\ T)$$

- select(DT)→LT

$$union_{merge}(select_{dist}(DT\ T))$$

- select(PT)→LT

$$union_{elim}(select_{dist}(DT\ T))$$

  or

$$union_{merge}(select_{dist}(PT\ T))$$

  or

$$union_{merge}(select_{dist}(partition(DT\ T)))$$

Sort

General sort algorithms require memory for all sorted tuples. To sort distributed tables while respecting our restrictive memory requirements, the $order_{dist}$ sorts only local table partitions in each node. We can use the $union_{merge}$ operator for merging the sorted sequences into a global result that is streamed to the root (query node).

Aggregation

Abstract global algebra aggregation has multiple concrete global algebra operations: $aggr_{local}$, $aggr_{dist}$, $aggr_{merge}$.

Aggregation can be processed by pipelining, if the input stream is already sorted on the groupby attributes. So, if there is an aggr in a query without previous sorting on its groupby attributes, the processor needs to add an $order_{dist}$ (for DT, PT) or $order_{local}$ (for LT) before the aggr.

We give translations of aggr(T) from abstract algebra to concrete algebra

- with elimination of replicated tuples:

  if table T is a partitioned table:

$$aggr_{merge}(aggr_{dist}(PT\ T))$$

  if T is a DT only, then we can still do:

$$aggr_{merge}(aggr_{dist}(partition(DT\ T)))$$

or

$$aggr_{local}(union_{elim}(DT\ T))$$

- without elimination:

$$aggr_{merge}(aggr_{dist}(DT\ T))$$

$$aggr_{local}(union_{merge}(DT\ T))$$

The choice depends on T being a DT or PT and the estimated number of groupby values.

Join

When joining two local tables, we can use $join_{local}$ directly. When joining a LT with DT or PT, we can use $union_x(join_{bcast}(LT, XT))$ with x=merge if XT=PT and x=elim if XT=DT.

The most difficult case is a join of two distributed tables A and B. Here we must make a join of the union of all local tables with schema A with union of all local tables with schema B.

$$A \bowtie B = (\cup(A_1, A_2, ..., A_n)) \bowtie (\cup(B_1, B_2, ..., B_n)) =$$

$$= A_1 \bowtie B_1 \cup A_1 \bowtie B_2 \cup ... \cup A_1 \bowtie B_n \cup A_2 \bowtie B_1 \cup ... \cup A_n \bowtie B_n$$

For $A \bowtie B$ it is necessary to join on all combinations of local tables A_i and B_j. But without transferring data, we can process only those combination A_i and B_j where $i = j$. It is necessary to transfer one table A_i or B_j to node where is another one or transfer both tables to one place and process partial join $A_i \bowtie B_j$ there.

We have the following alternatives to translate a global abstract join to concrete algebra:

- With $join_{local}$

$$join_{local}(union_{merge}(PT(T_1)), union_{merge}(PT(T_2)))$$

- With $join_{foreignkey}$, for foreign key join

$$union_{merge}(join_{foreignkey}(PT(T_1), PT(T_2)))$$

- With $join_{broadcast}$.

$$union_{merge}(join_{broadcast}(PT(T_1), LT(T_2)))$$

- $Join_{split}$ is similar to a $join_{broadcast}$.

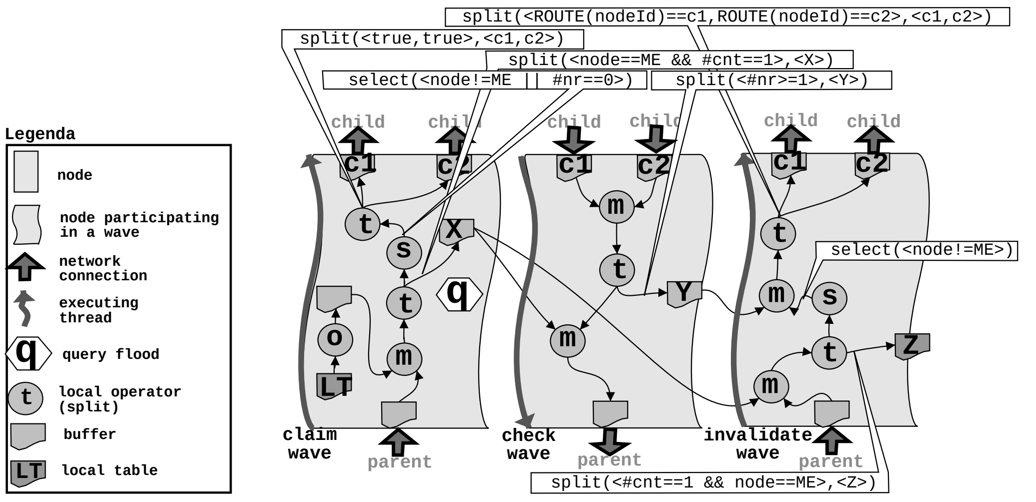

split(<ROUTE(nodeId)==c1,ROUTE(nodeId)==c2>,<c1,c2>)
split(<true,true>,<c1,c2>)
split(<node==ME && #cnt==1>,<X>)
select(<node!=ME || #nr==0>)
split(<#nr>=1>,<Y>)
Legenda
node
node participating in a wave
network connection
executing thread
query flood
local operator (split)
buffer
local table
child
c1
c2
t
s
X
q
o
m
LT
Y
Z
select(<node!=ME>)
claim wave
check wave
invalidate wave
parent
split(<#cnt==1 && node==ME>,<Z>)

Figure 3.4: The 3-wave Partition Operator

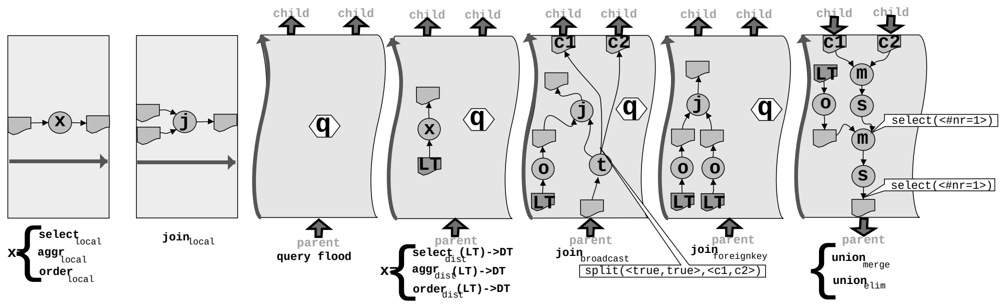

child
parent
x
j
q
o
LT
t
c1
c2
m
s
select(<#nr=1>)
select(<#nr=1>)
x= select_local, aggr_local, order_local
join_local
query flood
x= select_dist(LT)->DT, aggr_dist(LT)->DT, order_dist(LT)->DT
join_broadcast
split(<true,true>,<c1,c2>)
join_foreignkey
union_merge
union_elim

Figure 3.5: Wave Plans for most Concrete Operators

3.4.2. Translation from Concrete Algebra to Dataflow Operators

Concrete algebra operators are translated to a dataflow execution plan. We will discuss only partition in particular, other operations are defined in the figure 3.5. Local variants of operators are processed without communication on the query node(Figure3.5 a,b). other variants use usually the first wave for query propagation(Query flood in Figure 3.5 c). Operators producing DT or PT have only this one wave, the result is distributed over all nodes as DT or PT (Figure 3.5 d,f). $Join_{bcast}(LT, DT)$ has one wave. The table LT is broadcast in this wave to all nodes and the result is again distributed over all nodes as DT or LT. Union operators ($union_{merge}$ and $union_{elim}$ Figure 3.5 g) have two waves: query propagation and result merge.

The partition (Figure 3.4) is processed in three waves.

- CLAIM WAVE - down

  Each node receives a data stream from the parent node and merges with the ordered local table. The split makes from this stream two streams: one with all tuples and the other with tuples that come from the local table and cnt==1 (there are no equal tuples from the parent). The second data stream is stored in the buffer X. The first data stream is selected on the condition nr==0 (this condition makes the claim stream unique) and split into identical data streams, which are sent to the child nodes.

  Leaves: The first wave finishes in the leaves. Leaves have no child nodes, so execution is terminated by the *split* that creates the buffer X.

  Each node has a buffer X with tuples from LT which are candidates for participating in the new PT.

- CHECK WAVE - up

  When the Claim wave comes to the leaves, the second wave starts from the leaves to the root node. Leaves start the check wave by sending the content of the buffer X to their parent node.

  Each inner node first merges the data stream from child nodes and the result is split in two streams: one on the condition nr≥1 and the second output with all tuples where nr==0. The first data stream, which contains tuples that have been claimed twice, is written to buffer Y and the second stream is merged with the content of the buffer X and the result is sent to the parent node.

  The buffer Y contains information about tuples which after the Claim wave were in the buffer X, but in the Claim wave it was discovered that multiple nodes have claimed them. In the buffer Y we store also information about source nodes for each tuple (a NODEID field).

- INVALIDATE WAVE - down

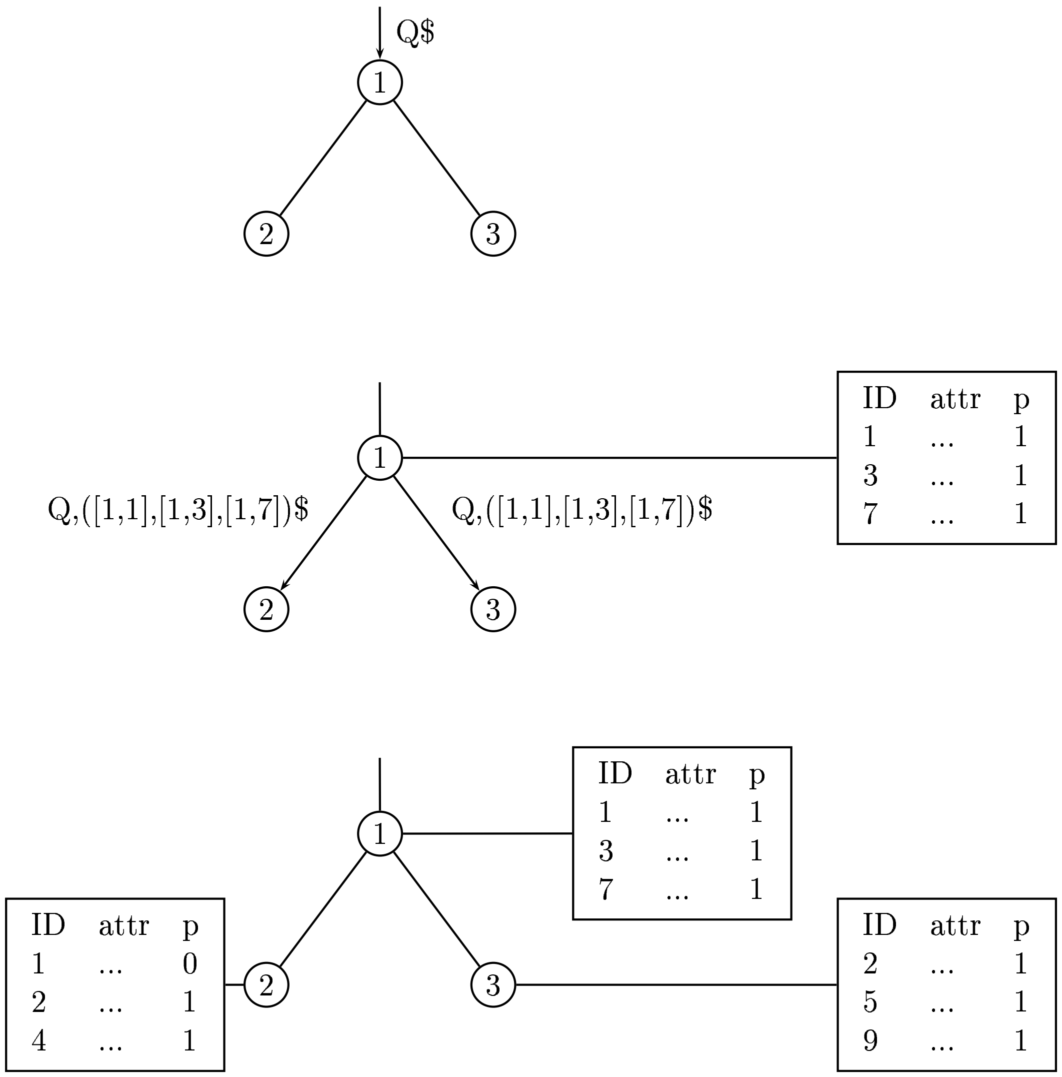
  This wave starts in the root node, which sends the content of the buffer Y to the child nodes. Each tuple from the buffer Y is routed to the source node N_i, where we create a new buffer Z_i

  $$Z_i = \{x; x \in X_i \wedge x \notin Y_j, \forall j \in Q\}$$

  Each node n_i merges the incoming data stream with its own buffer X_i and the result is split to two streams. One on the condition that tuple source is the local node and cnt==1. The second data stream on the condition that the tuple source is different from the local node. This stream is merged with the buffer Y_i and tuples are routed by split to the corresponding child nodes. The first stream is written to buffer Z_i, which makes up the new partitioned table, after the invalidation wave has finished.

Implementation notes

- It is not necessary to send all tuples, but only key values.
- Data that are sent in the Claim wave and the Check wave are sorted by the key attribute.
- The Check wave and the Invalidate wave can start before the preceding wave is finished.
- Instead of materializing Z, we can make a bit in a bitmap column of LT to identify which tuples participate in the PT with low storage overhead.
- The NODEID should be implemented in such a way that ROUTE(id) can be efficiently determined. One solution is to use concatenations of child numbers as NODEIDs.
- Size of the buffer X and the buffer Z is limited, because it contains less or equal number of items as local table, but the size of the buffer Y is not limited. The invalidation wave can start before previous waves are finished. The content of the buffer Y is merged with the incoming stream in the described implementation of partition, but it can be sent to child nodes before. The invalidation wave comes to the node.



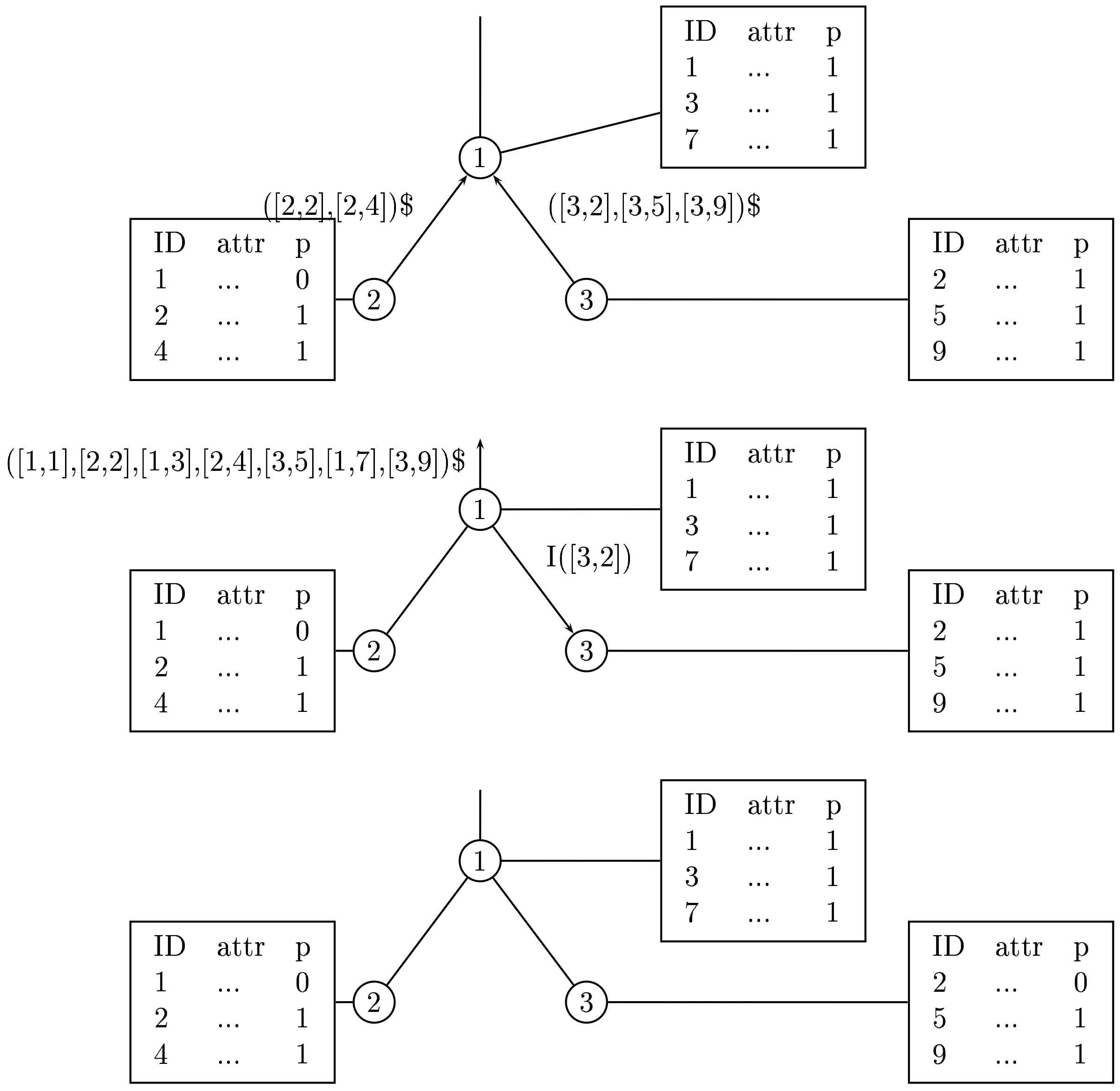
ID	attr	p
1	...	1
3	...	1
7	...	1

ID	attr	p
1	...	1
3	...	1
7	...	1

ID	attr	p
1	...	0
2	...	1
4	...	1

ID	attr	p
2	...	1
5	...	1
9	...	1

Figure 3.6: Partition - Claim wave



ID	attr	p
1	...	1
3	...	1
7	...	1

ID	attr	p
1	...	0
2	...	1
4	...	1

ID	attr	p
2	...	1
5	...	1
9	...	1

ID	attr	p
1	...	1
3	...	1
7	...	1

ID	attr	p
1	...	0
2	...	1
4	...	1

ID	attr	p
2	...	1
5	...	1
9	...	1

ID	attr	p
1	...	1
3	...	1
7	...	1

ID	attr	p
1	...	0
2	...	1
4	...	1

ID	attr	p
2	...	0
5	...	1
9	...	1

Figure 3.7: Partition - Check wave

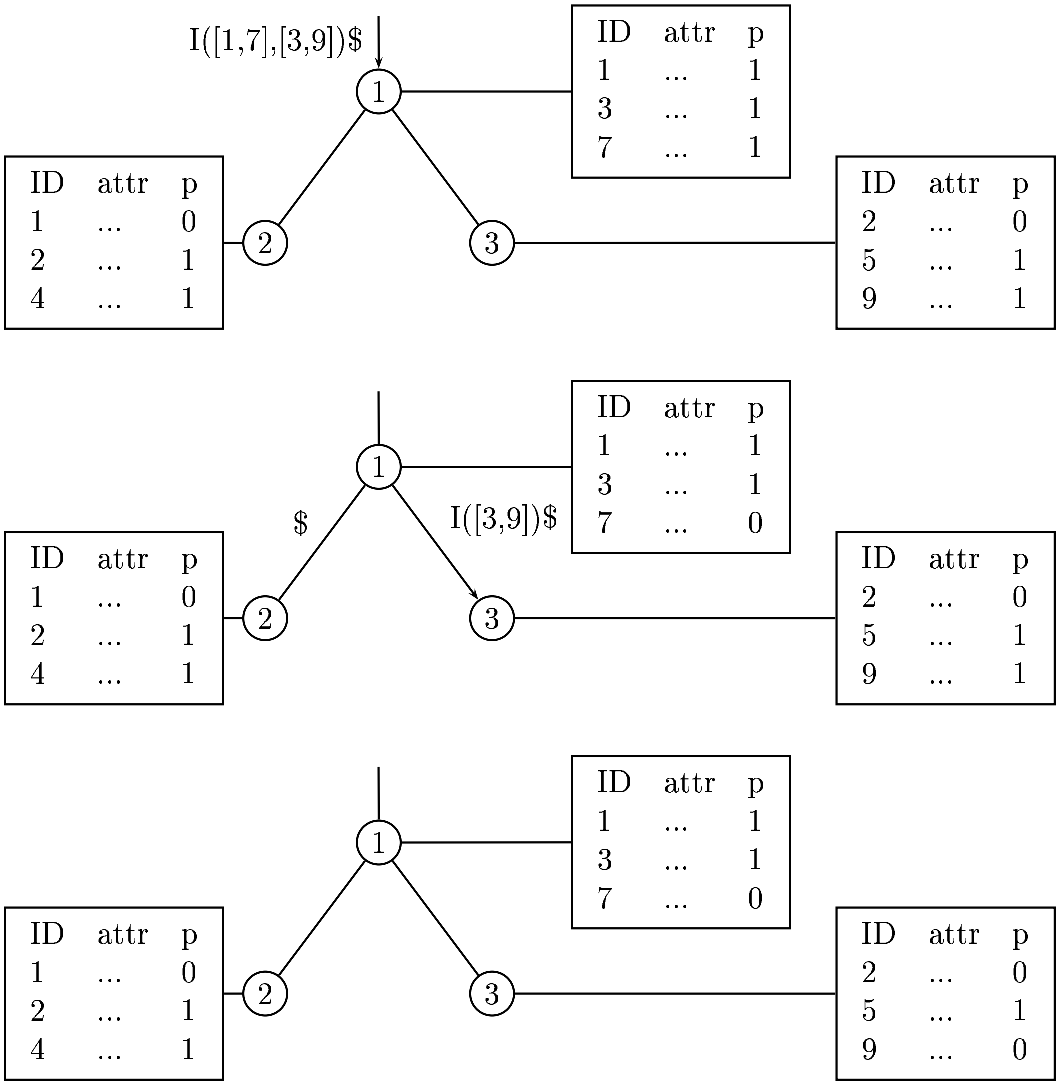


Figure 3.8: Partition - Invalidation wave

Chapter 4

Implementation

In order to obtain insight into the correctness and performance of our QP framework we decided to implement a simulator. Assumptions allows to easily task the framework on a wide variety of network configurations and load. In this simulation, we only obtain network costs for queries, which provides efficient feedback as we assume network costs to be dominant. In order to reduce the work of creating this simulation, we decided to implement a number of hardcoded queries rather than a general-purpose query interpreter.

4.1. Network Simulator ns2

ns2 [7, 1] is a discrete event simulator targeted to networking research. ns2 provides support for simulation of TCP, routing, and multicast protocols over wired and wireless networks.

ns2 is an object-oriented simulator written in C++. Its front-end is OTcl inter-

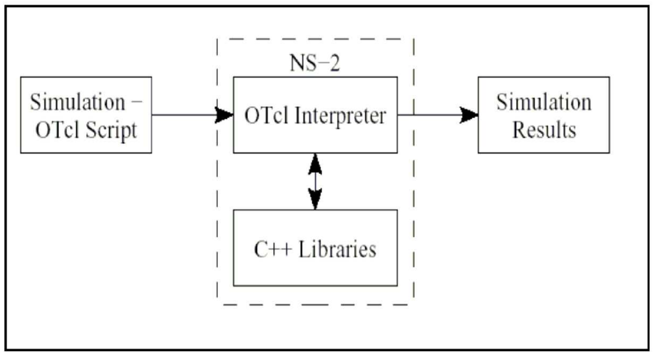


Figure 4.1: Network simulator ns2

preter. The simulator consists of class a hierarchy in C++ and OTcl and there is one-to-one correspondence between them. Clients create new simulator objects in the interpreter. ns2 uses these two languages (C++ and OTcl) for two different tasks: OTcl is used for configuration and setup. C++ is used for processing each packet of the flow and changing the behavior of the existing C++ class. It is possible to invoke OTcl procedures from C++ code and vice versa.

4.2. Data Structures

4.2.1. Agents

Agents represent endpoints at which network-layer packets are constructed and consumed. The `class Agent` is implemented partially in OTcl and partially in C++. It supports packet generation and receiving and following functions are implement by C++ Agent class:

`void recv(Packet*, Handler*)` is the main entry point of the Agent which receives a packet. It is invoked by upstream nodes when sending a packet.

The `Agent::allockpkt()` method allocates a new packet, fills in its common and IP headers. Appropriate TCP-layers headers fields have to be filled in them.

`class PartAgent` is a subclass of standard build-in `class Agent`. Data structures and methods of the `class PartAgent` are described later. Each node of our tree consists of 1 or more agents. The agent which is connected to the parent node (with 0 for root node) is denoted as main. This main agent contains data and information about the query processing state. Other agents are used only for sending and receiving packets.

We use a synthetic database in this simulation, which we now fill with a uniformly distributed integer values. These values are stored in a linked-list. Each data_item contains a key value, a partition flag(used in the partition operator to denote of a tuple of a DT participates in a PT) and a pointer to the next data_item.

Each main agent also contains information about its child nodes. This information is again stored in a link-list of child_items. Each child_item contains a pointer to the child agent and a link-list with received data items.

4.2.2. Tree Topology

The tree topology is defined and initialized in OTcl code. Thus, network topology can be generated as a TCL script. C++ methods are called during the initialization and they initialize the same topology on C++ objects. The OTcl part of the code defines nodes and connections between nodes. Each node of the tree is composed of 1 or more agents. Each agent is used for exactly one connection. Leaf nodes have only

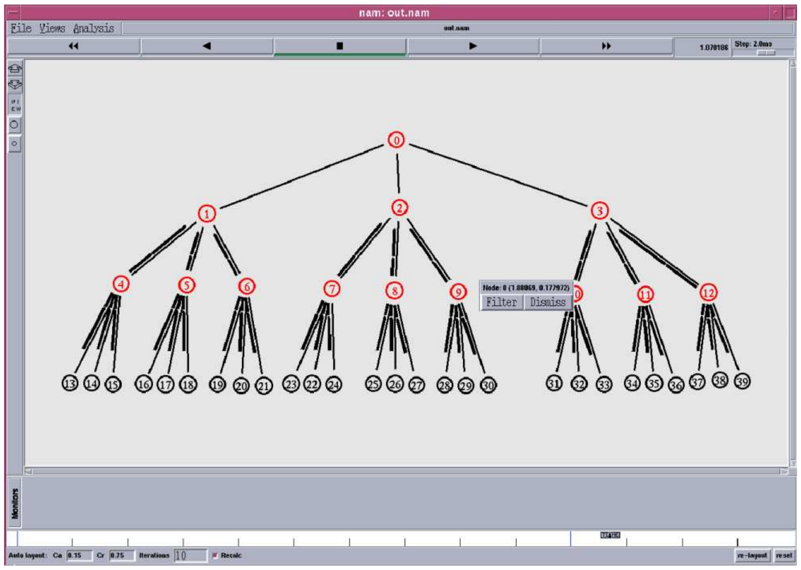


Figure 4.2: Netowork animator - nam

one agent for the connection with their parent node. Inner nodes have one agent for the connection with their parent node and one agent for each child node.

4.2.3. Packet

Each sent packet has the same structure. The size of the packet is defined in file ns-default.tcl. There is method Agent::allockpkt() for allocation of a new packet. This method allocates a new packet and fills its IP headers. The access() method returns a pointer to the TCP headers that should be filled. This part of the packet is used to carry data, because ns2 does not support any other way to transfer real data.

4.3. Communication

PartAgent is inherited from Agent. It uses two Agent methods: *send* and *recv*. These two methods are modified as follows:

We have a method newSend which calls the original method send. This method also prepares data items to a packet and sends the packet, when it is full or at the end of the wave (marked by special symbol).

The method recv parses data items carried by the packet and calls a method for processing the data items. It decides which method to use for processing according to agent's position in the tree, direction of the wave, and value of the data item.

4.4. Algorithm

Commands, which are programmed in C++ and used in OTcl code, are parsed by the command method. Arguments of command are stored in an array. Following hardcoded operations are implemented: partition, select, aggregation and join. These operations work with partitioned or distributed tables. There we again only describe partition in detail.

4.4.1. Partition

The methods are illustrated at figure 4.3.

1.wave All nodes, except for leaf nodes, use the mergeOwnDown method for processing. It merges the incoming stream of data items with its own set of data items. This method also marks local items, in case that the same key value occurs also in the incoming stream.

Method distrSendDown is used for sending items to child nodes. It sends data item to all child nodes, and calls method newSend for each child node.

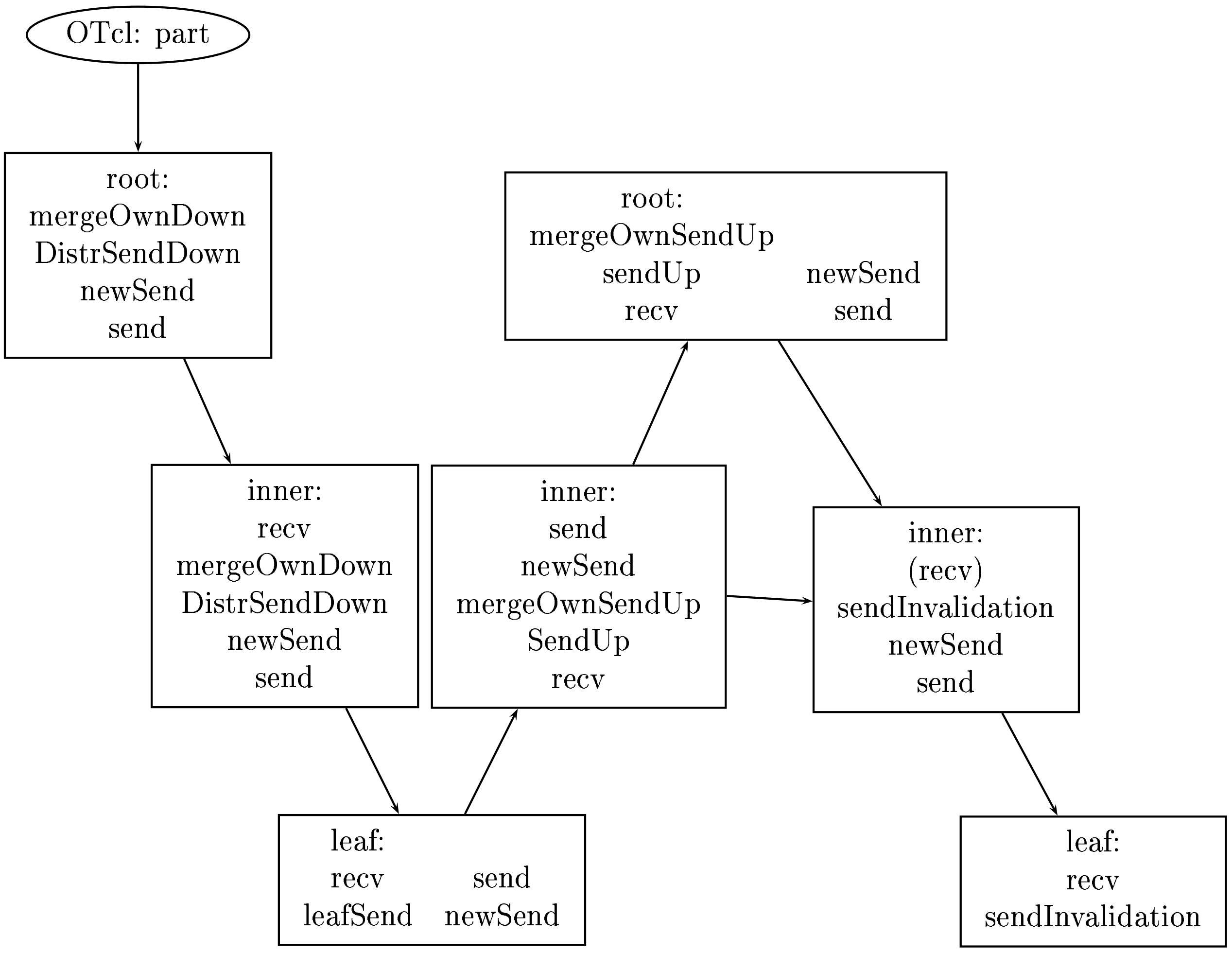
The direction of the wave is changed in the leaf nodes. So, the method leafSend is used in leaf nodes. It marks local replicated data items and starts a new wave in upward direction. It sends its own data items which are not marked as replicated in this new wave.

The method sendInvalidation sends invalidation requests, which were generated during the upward wave. This method finds in which branch the receiver of the invalidation is and sends this request to the child node, which is the root node of that branch.

2.wave The sendUp method is called in method recv during the wave upward for all agents. This method is not called for the receiving agent but for main PartAgent of the same node. The sendUp method adds this item to the end of the linked-list which belongs to the calling agent. Then it starts taking the minimum items of all linked-lists of received data from all child nodes. It stops when at least one linked-list is empty. When the same item occurs in more than one linked-list then it generates an invalidation request for all such nodes except the first one. The Invalidation request is generated by calling the method sendInvalidation. The items are merged with data items stored in the node by method mergeOwnSendUp.

The mergeOwnSendUp method merges its own non-replicated items with incoming data streams from all children and sends the result using the newSend method to the parent node.

Figure 4.3: Schema of simulation of operation partition



3.wave Invalidation requests received in packets or directly from function sendInvalidation are routed to the corresponding nodes again by function sendinvalidation. This function also marks corresponding items in local table if the invalidation request belongs to the node.

Chapter 5

Evaluation

5.1. Scenario

As the P2P system has up to now been an academic exercise only, we formulate a number of usage scenarios where such a P2P DBMS might be used in the future. In each scenario, we define the system parameters and a set of queries to allow conducting a number of experiments.

For each scenario description we will deduce a typical network size and physical characteristics of the network, nodes and data.

Each node can be described by size of memory, size of RAM, processor and network connection. We can also describe data stored in the node: size of data stored in the node and which schemata are not used in the node.

Scenario "Household" In this scenario we will assume household with common home devices. There will be approximately tens of devices. The devices can be divided to groups according to hardware properties:

Size	Devices	Memory	Connection
big	Computer, laptop, ...	10-50GB	
medium	PDA, mobile phone,digital camera, ...	2-100MB	
small	refrigerator, central heating, ...	...	
very small	light sensors, temperature sensors, ...	...	

Global data schema

```
tables:
Addressbook(ID_addr, name, nickname, phone_number, addr_street,
addr_town, addr_Post_nmr)
Artist(ID_artist, name, country, style)
Album(ID_album, ID_art, name, year, style)
```

```
Songs(ID_song, name, style, ID_album, ID_artist,lenght)
Common_resources(Id_res, name, amount)
Organiser(Id, action, type, description, time_begin, time_end, place)
logs:
log_phone_calls(phone_number, time, duration)
log_played_songs(time,ID_song,volume, response, ID_user)
log_resources(ID_res, amount, buy_date, exp_date)
log_motion(time, room, motion)
log_lights(room, ID_light, intensity, time_begin, time_end)
log_temperature(room, time, temperature)
log_heating(room, time, intensity)
...
```

Data extracting

- Some data items are created consciously as new meeting in Organiser or new address in Addressbook.
- Some data items are created after some action. Such as when the light being turn on then is written into log that light was turn on in time t.
- Some data items are created periodically. For example, every minute a value of a temperature sensor is generated.

Schema on devices

Computer, laptop	All
Refrigerator	log_resources, Common_resources
mobile phone, phone	Addressbook, log_phone_calls
PDA	Addressbook, Organiser
PM3-player, stereo	Song, log_played_songs

Queries

- **SQL query** Create list of songs from Beatles

  ```
  SELECT DISTINCT S.name
  FROM Songs S, Artist A
  WHERE S.ID_art=A.ID_art AND A.name='Beatles'
  ORDER BY S.name
  ```

  Outline of execution plan

$$union_{merge}(select_{dist}(join_{foreignkey}(PT\ S,\ DT\ A)))$$

or

$$union_{elim}(select_{dist}(join_{foreignkey}(DT\ S,\ DT\ A)))$$

Network simulation

$$union_{merge}(PT)$$

or

$$union_{elim}(DT)$$

- **SQL query** Give me a list with people and total time phoned with them, sorted by the most often to the least.

  ```
  SELECT A.name, sum(L.duration)
  FROM Addressbook A, log_phone_calls L
  WHERE A.phone_number=L.phone_number
  GROUP BY A.name
  ORDER BY sum(L.duration)
  ```

  Outline of executing plan

  $$order_{local}(aggr_{merge}(join_{foreignkey}(DT\ A,\ PT\ L)))$$

  or

  $$order_{local}(aggr_{local}(union_{elim}(join_{foreignkey}(DT\ A,\ DT\ L))))$$

  Network simulation

  $$aggr_{merge}(PT)$$

  or

  $$union_{elim}(DT)$$

- **SQL query** Lets assume that we have table Relevant with all users and parameter of similarity of our music preferences. We want to make list of songs ordered by my expected preferences.

  ```
  SELECT A.name, S.name, sum(R.parameter*L.response)
  FROM Relevant R, log_played_songs L, Songs S, Artist A
  WHERE R.ID_user=L.ID_user AND L.Id_song=S.Id_song
  AND S.ID_artist=A.ID_artist
  GROUP BY L.ID_song
  ORDER BY sum(R.parameter*L.response)
  LIMIT 100
  ```

  Outline of executing plan

  $$order_{local}(aggr_{merge}(join_{bcast}(LT\ R,\ PT\ JL)))$$

  where

  $$JL := join_{foreignkey}(PT\ L, join_{foreignkey}(DT\ S,\ DT\ A))$$

Network simulation

$$aggr_{merge}(join_{bcast}(LT, PT))$$

5.2. Benchmarks

For each scenario, a set of typical queries and data operations is given. For each set of queries and each data distribution another query execution strategy can be better. For comparing individual execution strategies we can change the following input properties:

- number of nodes
- network topology
- query set
- data distribution among nodes
- ratio of replicated data

ns2 allows measure only communication costs:

- number of messages (or bytes)
- total communication time

5.3. Results

Abstract global algebra operators are translated to concrete global algebra operators. Only some of them use "in network" processing.

Local versions of operators are processed locally in the query node without communication costs.

Distributed versions of operators need communication only for query propagation to all nodes.

Operators processed "in network" are the only interesting ones for measuring communication costs. So the following operators are discussed: $partition$, $union_{merge}$, $union_{elim}$, $aggr_{merge}$ and $join_{bcast}$. In our experiment, we were measuring the number of messages and the total communication time.

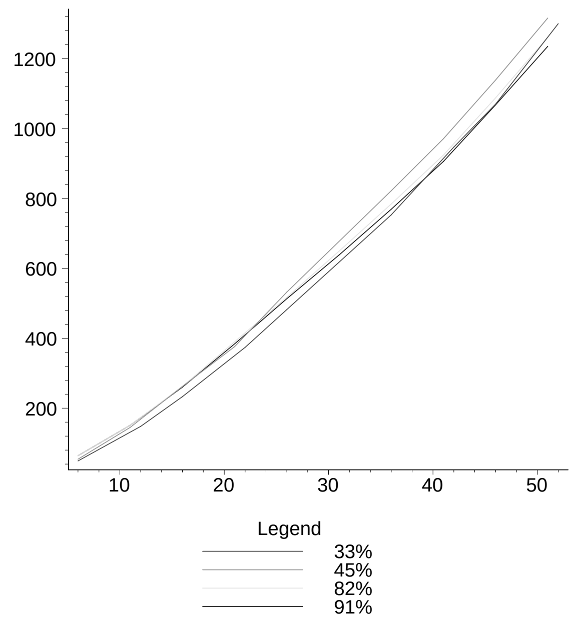


Figure 5.1: Dependence of total number of messages on the number of nodes
All nodes are the same, with 1000 tuples in LT.

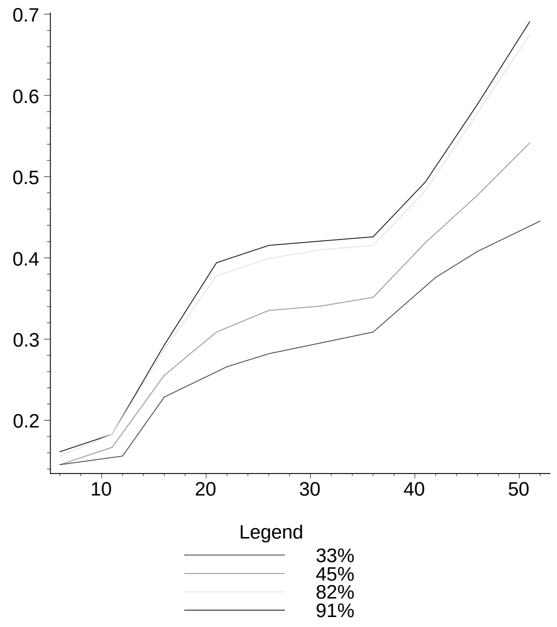


Figure 5.2: Dependence of total communication time on the number of nodes
All nodes are the same, with 1000 tuples in LT.

5.3.1. Partition

We measured the relation between communication costs and the percentage of unique items and the total number of nodes. All nodes were the same. The first graph (Figure 5.1) shows that the total number of messages seems to depend linearly on the number of nodes.

$$msg = O(n)$$

There are four curves, each for a different percentage of replication. In the next graph (Figure 5.2), there is a dependence of the total communication time on the number of nodes, again for four different replication percentages.

$$total\_time = O(\lceil f(log_3(n)\rceil))$$

for well-balanced graph where each inner node has three child nodes. The total communication time depends on the depth of the tree.

5.3.2. Select

Two basic variants of the translation of abstract global algebra operation select to concrete global algebra are proposed in chapter Design. We made measurements to compare $union_{elim}(select_{dist}(DT)) \rightarrow LT$ and $union_{merge}(select_{dist}(PT)) \rightarrow LT$. These experiment compared these two variants and their dependence on the percentage of unique items in distributed table.

The first graph (Figure 5.3) represents the relation of the number of messages on the percentage of unique items. In the worst case, when there are no duplicated items, $union_{merge}(PT)$ behaves the same as $union_{elim}(DT)$. In other cases, $union_{merge}(PT)$ has less communication costs. The difference between these two strategies is more obvious when we compare the number of messages rather than the total communication time (Figure 5.5).

5.3.3. Aggregation

We were comparing two variants for aggregation: First variant is $aggr_{local}(union_{elim}(DT))$ and the second is $aggr_{merge}(aggr_{dist}(PT))$. We used a database with 100% of unique items, $|DT| = |PT|$. We measured the relation between the number of messages and the size of the groupby set. In the first variant the result does not depend on the size of the groupby set. In the second variant the result depends on the size of the groupby set and the shape of the curve is similar to the $union_{elim}(DT)$ in graph Figure **??**.

5.3.4. Join

We implemented only $union_{merge}(join_{bcast}(LT, PT))$, which is from communication costs viewpoint the same as $aggr_{merge}(join_{bcast}(LT, PT))$

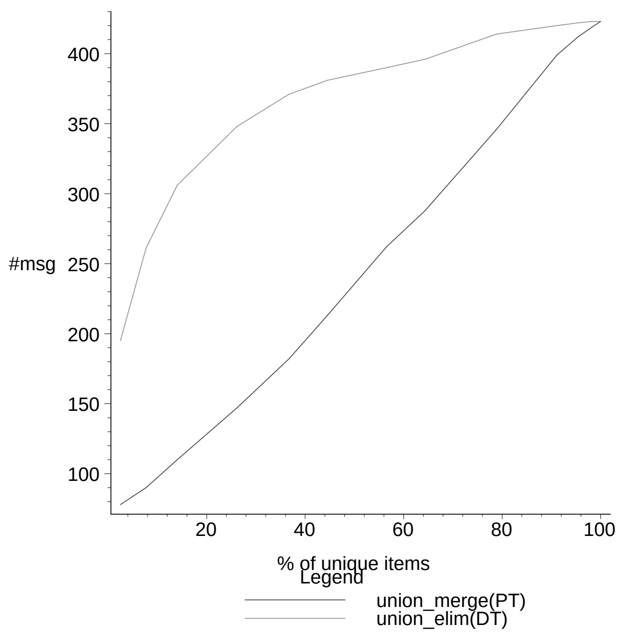


Figure 5.3: Dependence of the number of messages on the percentage of unique tuples

$join_{local}(LT)$ and $union_{merge}(join_{foreignkey}(DT))$ are not interesting from the viewpoint of communication. We measured the relation between the number of messages and the size of table LT. In Figure 5.7, there is a relation between the size of LT and the total number of messages. The result depends linear on the size of LT.

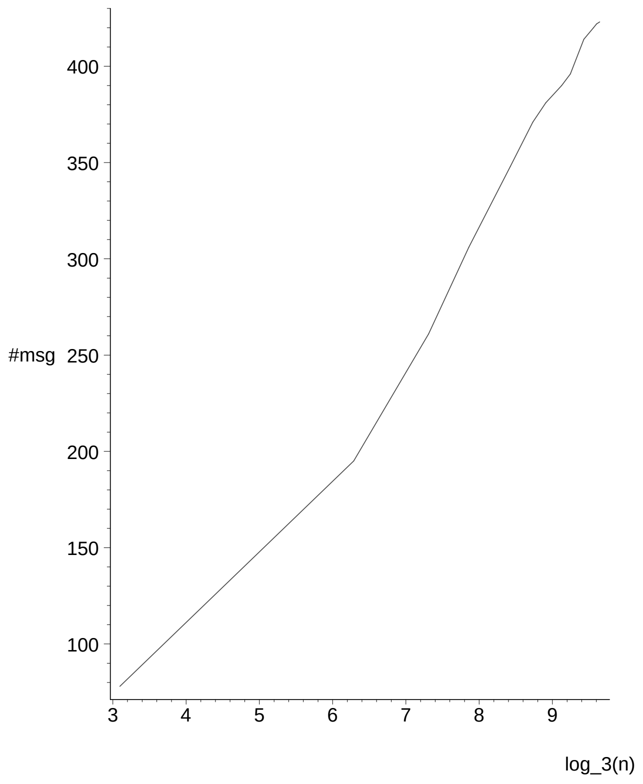


Figure 5.4: $Union_{merge}$: Dependence of total number of messages on $log_3(n)$, where n is total number of unique items

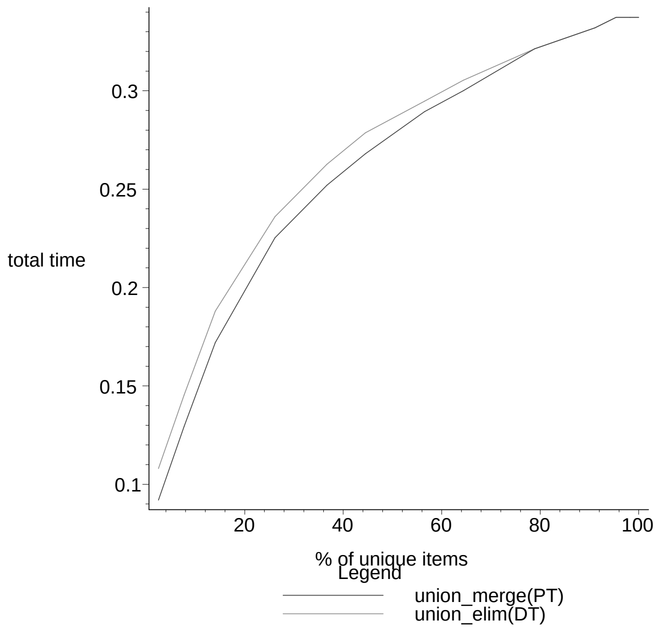


Figure 5.5: Dependence of time on the percentage of unique tuples

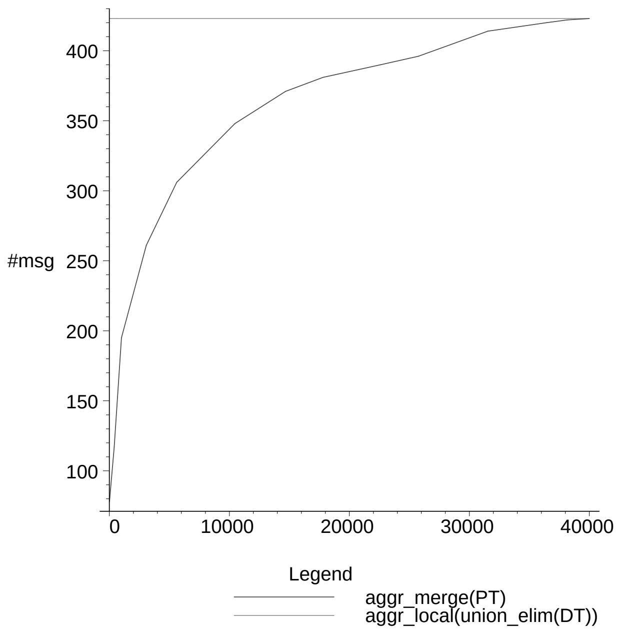


Figure 5.6: Dependence of the number of messages on the size of the groupby set

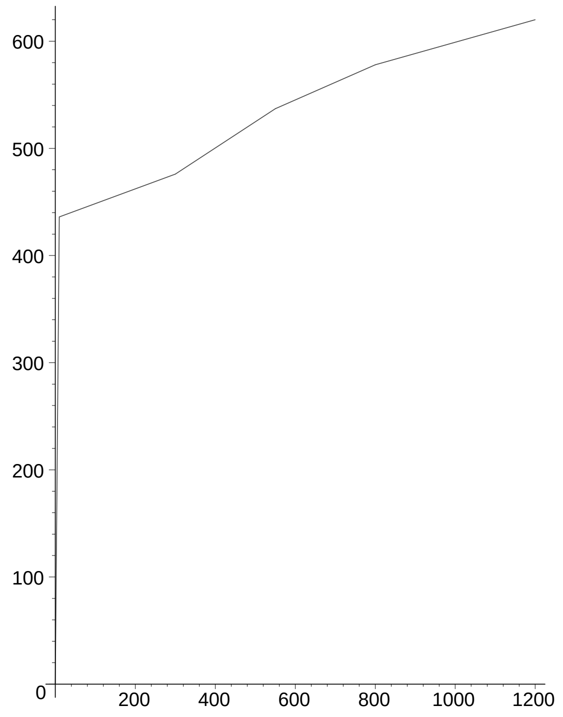


Figure 5.7: Dependence of the total number of messages on the size of the broadcast table.

Chapter 6

Conclusion

In this M.Sc. thesis, we have formulated a generic query processing framework for peer-to-peer databases. Thus, we think it is feasible and promising to bring complex query functionality to P2P systems.

A prototype on network simulator ns2 was implemented. Experiments on this prototype confirmed that algorithm is usable and correct. Experimental results on data with uniform distribution correspond to what we would expect.

As such, we regard this prototype as a tool and starting point for future experiments and research. Testing non-uniform input data or real data inputs could be the next step in the examination of the proposed query processing strategy. Also, new algorithms for efficient P2P query processing could be intended in our framework and form an additional topic of future research.

Acknowledgements

sdasd sadasd dasdas dsad

Bibliography

[1] Eitan Altman and Tania Jiménez. *NS Simulator for beginners*. World Wide Web, http://www_sop.inria.fr/mistral/personnel/Eitan.Altaman/COURS-NS/n2.ps.

[2] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data management for peer-to-peer computing: A vision. In *Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, june 2002.

[3] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez. Picobdms: Scaling down database techniques for the smartcard. In *Proceedings of the 26th International Conference on Very Large Databases*, Cairo, Egypt, 2000.

[4] Peter Boncz and Casper Treihtel. Ambientdb: Relational query processing in a p2p network. Technical Report INS0306, Centrum voor Wiskunde en Informatica, Kruislaan 413, Amsterdam, 2003.

[5] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. In *IEEE Personal Communications, 7(5)*, pages 10–15, October 2000.

[6] David J. DeWitt, Goetz Graefe, Krishna B. Kumar, Robert H. Gerber, Michael L. Heytens, and M.Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the Twelfth International Conference on Very large Data Bases*, Kyoto, August 1986.

[7] Kevin Fall and Kannan Varadhan. *The ns Manual*. World Wide Web, http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.

[8] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can peer-to-peer do for databases, and vice versa? In *Fourth International Workshop on the Web and Databases (WebDB'2001)*, Santa Barbara, CA, USA, May 2001.

[9] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication.

In *International Workshop on Mobile Data Access Workshop (ER'98)*, pages 254–265, 1998.

[10] Alon Y. Halevy, Zachary G. Ives, Dan Staciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *International Conference on Data Engineering*, March 2003.

[11] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *International workshop on Peer-to-Peer systems(IPTPS '02)*, Cambridge, MA, USA, March 2002.

[12] Martin L. Kersten and Arno P.J.M. Siebes. An organic database system. Technical Report INS-R9905, Centrum voor Wiskunde en Informatica, Kruislaan 413, Amsterdam, 1999.

[13] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[14] Samuel Madden, Michael J.Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[15] Peter Mc.Brien and Alexandra Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proceedings of the CAiSE02*, volume TBC. Springer Verlag LNCS, 2002.

[16] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall International, Inc., 1991.

[17] Annita N. Wilschut and Peter M.G.Apers. Dataflow query execution in a parallel main-memory enviroment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[18] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *The VLDB Journal*, pages 261–570, September 2001.