Bachelor Thesis

# Communication Efficient Algorithms for Distributed OLAP Query Execution

Demian Hespe

Oct 28, 2014

Supervisor: Prof. Dr. Peter Sanders
Dipl. Inform. Jonathan Dees
M.Sc. Martin Weidner

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 28. Oktober 2014

**Zusammenfassung**

In der heutigen Zeit steigen Datenmengen rasant an. In Folge dessen reicht eine einzelne Maschine häufig nicht mehr aus um diese zu speichern und zu verarbeiten. Daher wird in dieser Arbeit der Ansatz verfolgt eine Datenbank in einem verteilten System zu implementieren. Vor allem bei komplizierten analytischen Anfragen an die Datenbank fällt dabei allerdings häufig ein großer Anteil der Ausführungszeit auf die Kommunikation zwischen den einzelnen Maschinen des Systems ab, weshalb versucht wird das Volumen der kommunizierten Daten zu minimieren.

Zunächst werden frühere Arbeiten zu den Themen hauptspeicherbasierte und spaltenorientierte Datenbanken, kommunikationseffiziente Gruppenkommunikationsalgorithmen und verteilte Datenbanken betrachtet, deren Ergebnisse zum Teil für die in dieser Arbeit entwickelten Algorithmen verwendet werden oder die Grundlage für eine Komplexitätsanalyse bilden.

Im Hauptteil werden Techniken, die bereits in einer früheren Arbeit, auf der diese Arbeit aufbaut, entwickelt wurden, analysiert und Verbesserungen dieser vorgeschlagen. Es werden hierbei im speziellen Algorithmen für Anfragen betrachtet, die nur die ersten $k$ Tupel des Ergebnisses fordern und die nicht allein durch lokale Berechnungen auf den einzelnen Recheneinheiten der verteilten Datenbank gelöst werden können und daher einen großen Kommunikationsaufwand bergen. Es werden hier große Optimierungsmöglichkeiten erwartet, da diese in den Skalierungsexperimenten der vorhergehenden Arbeit keine zufriedenstellenden Eigenschaften aufwiesen.

Anschließend werden drei ausgewählte Anfragen aus dem *Online Analytical Processing* (OLAP) Benchmark TPC-H genauer analysiert und mit Kommunikationsaufwand verbundene Joins in diesen identifiziert. Zur effizienten Ausführung von Joins, werden die zuvor entwickelten Algorithmen auf diese Anfragen angewandt und gegen andere Lösungsmöglichkeiten abgewogen um die Entscheidungen zu begründen.

Zur Evaluierung der neu entwickelten Algorithmen werden die drei Anfragen des TPC-H Benchmarks in Form eines Prototyps implementiert und in einem großen Rechnerverbund von 128 Knoten auf einer Datenbank mit bis zu 30 Terabyte (zum Teil bis zu 128 Terabyte) an umkompressierten Daten ausgeführt. Hierbei werden zunächst Skalierungsexperimente durchgeführt um den Einfluss der Anzahl der Knoten des Rechnerverbundes auf die Laufzeit und den Kommunikationsanteil derselben zu analysieren. Anschließend werden die gemessenen Laufzeiten mit denen der vorhergehenden Arbeit und des aktuellen TPC-H Rekordhalters verglichen. Dabei werden im Vergleich zum aktuellen Rekordhalter bis zu 35 mal schnellere Laufzeiten erreicht und im Vergleich zu der früheren Arbeit deutlich bessere Skalierungseigenschaften, die zum Großteil auf deutlich geringere Kommunikationsvolumen zurückzuführen sind, erkannt. Diese resultieren auch in schnelleren Laufzeiten, wenn die Algorithmen auf vielen Knoten im Cluster und großen Datenbeständen ausgeführt werden.

Der implementierte Prototyp wird mit Hilfe performanter Bibliotheken

zur Parallelisierung implementiert und verwendet Implementierungen des offenen Standards Message Passing Interface (MPI). Um schnelle Ausführungszeiten zu gewähren, werden alle Daten zu jeder Zeit im Hauptspeicher gehalten und Spaltenweise gespeichert, da dies sich als effizient für analytische Anfragen erwiesen hat. Zur weiteren Verbesserung der Laufzeiten wurden einzelne Teile von *OpenMPI* durch alternative Implementierungen ersetzt, die in durchgeführten Tests bessere Ergebnisse erzielten.

Die Hauptbeiträge dieser Arbeit sind:

- Eine Technik um bessere Partitionierungen der Tabellen einer Datenbank zu finden, die es ermöglichen einige Joins ohne zusätzliche Kommunikation auszuführen
- Ein Algorithmus, der zur Auswahl der ersten $k$ Elemente des Ergebnisses einer Anfrage bei bestimmten Partitionierungen der Datenbank ein von der Größe der Datenbank unabhängiges Kommunikationsvolumen verursacht, indem er die nicht lokal auswertbaren Filterbedingungen der Anfrage faul auswertet (lazy evaluation)
- Eine Analyse des Kommunikationsbedarfs beim Verzögern eines Joins, der nicht lokal auf einer Maschine ausgewertet werden kann, und ein Vergleich zum Kommunikationsaufwand bei einer frühen Durchführung des Joins. Dazu werden bewiesene untere Schranken der benötigten Gruppenkommunikationsprimitive benutzt um aussagekräftige Ergebnisse zu liefern
- Die Anwendung der entwickelten Algorithmen auf komplexe Anfragen des TPC-H Benchmarks, die nicht ohne höheren Kommunikationsaufwand zu lösen sind
- Die Implementierung der entwickelten Algorithmen in einem Prototyp einer verteilten Datenbank und die Evaluation desselben in einem Cluster von 128 Knoten auf einer Datenbank von bis zu 30 (oder 128, falls nur ein kleiner Teil der Datenbank genutzt wird) Terabyte an unkompressierten Daten

Außerdem werden bewährte Techniken aus der vorhergehenden Arbeit übernommen, wie eine Kommunikationsvolumensparende Möglichkeit aus den Teilergebnissen der einzelnen Knoten ein Gesamtergebnis des Rechnerverbundes zu ermitteln, wenn nur die ersten $k$ Tupel des Ergebnisses benötigt werden. Des Weiteren wird auch erklärt, wie Kommunikationsvolumen durch das Ausnutzen früher kommunizierter Informationen eingespart werden kann, was die Laufzeiten erneut senkt.

Den Abschluss der Ausarbeitung bildet ein Rückblick auf die erbrachten Leistungen sowie eine Analyse möglicher weiterer Arbeiten, die nötig sind um die entwickelten Algorithmen in ein Marktreifes System zu integrieren.

**Abstract**

As a result of the growing amounts of Data in todays Databases, one machine is often not sufficient to store and process these. The proper solution to this problem is to scale the system out on a cluster. However, the distribution of the data throughout the machines of the cluster results in a high percentage of communication time in the overall execution time of a query, especially for complex analytical queries. For this reason, we try to minimize the volume of communicated data to allow faster runtimes when a query cannot be executed on a single node of the cluster without any communication. We analyze techniques from previous work and propose improvements to them backed by a complexity analysis of the communication volume for both, our algorithms and the algorithms from the previous work.
For the evaluation of our algorithms we implement them for chosen queries of the TPC-H benchmark and run them on a cluster of up to 128 nodes with a database of up to 30 terabytes of uncompressed data (128 TB if only a small proportion of the database is used). We provide both, scaling experiments and runtime comparisons to previous work and the current TPC-H record holder.

The main contributions of this work are:

- A technique to find a better partitioning of the tables in a database to allow the execution of joins without communication effort
- An algorithm that selects the first $k$ tuples of the result set of a query with a communication effort independent from the size of the database, given certain conditions of the partitioning
- An analysis of the communication effort of a delayed join that can't be evaluated locally on a node, in comparison to the communication effort when executing the join early
- The application of our algorithms to solve complex queries of the TPC-H benchmark that can't be executed without a high amount of communication effort
- The implementation of the queries in a prototype and evaluation of our algorithms on a large cluster consisting of 128 nodes for a database with up to 30 terabytes of uncompressed data (or 128 TB if only a small proportion of the database is used)

# Contents

# List of Figures



# List of Tables

# 1. Introduction

In todays big data warehouses fast online analytical processing (OLAP) becomes more and more important in order to avoid long waiting times for ad-hoc queries. While the speed improvement of hardware becomes slower over time, the amount of data to be processed increases drastically. To be able to process such large data volumes parallelization becomes essential. As the number of processing units and memory on a single machine is usually not large enough and the prices for high performance machines are getting higher, scaling systems up does not seem like a solution for this problem. However, the possibility of scaling a system out to multiple machines is a promising approach as those single machines are cheaper than one big machine and there is virtually no bound on the number of computers that can be combined into a cluster.
Despite the huge combined performance and memory of a cluster a new bottleneck evolves in the linking network [19], thus developing more communication efficient algorithms for distributed query execution is required.
As there was a lot of research on the area of sophisticated group communication algorithms and there are widely available implementations of these, like the Message Passing Interface (MPI), we will base our algorithms on the usage of group communications to achieve the smallest possible amount of communication effort.

## 1.1. Contribution

In this work we analyze common patterns that appear in many OLAP queries and that require vast amounts of communication when executed on a distributed database.
We provide solutions for queries following these patterns that are very efficient in terms of the data transmitted between the nodes of the cluster running the database using group communication operations. In order to support decision making between different approaches to minimize communication, we analyze our algorithms using proven lower bounds for common group communication primitives and compare these results with other sophisticated solutions.

In particular, for certain data distributions, we develop an algorithm to select the top $k$ results of a query with a communication effort independent from the size of the total result set, even if the query contains a filter condition that can't be evaluated locally.
We also prove that, in most cases, it is beneficial in terms of communication volume to perform joins, that can't be evaluated locally on a node, as late as possible.

Another contribution is a generalized way to find better distributions of the data that enable the database to evaluate most joins locally without the need of communication while executing the query.

To evaluate the performance of our algorithms in terms of practicality for real-world applications, we combine our ideas to run chosen queries from the TPC-H benchmark that seem to be hard to execute efficiently as other work shows. We implement these queries in a prototype of a distributed database, run them on a large cluster with an input of up to 30 terabytes of uncompressed data (128 terabyte for queries that use

only a small proportion of the input data) and compare our results to previous work and the current record holder of the TPC-H benchmark, where we achieve runtimes of factor 11 to 35 faster. We also run both, strong and weak scaling experiments to evaluate the behavior of our algorithms when scaling out the system and achieve a good scaling behavior especially for weak scaling experiments which is the kind of scaling out systems we expect in real-world applications.

We achieve these fast runtimes by making full use of the available hardware, state-of-the-art parallelization libraries and sophisticated algorithms from other work: We use full multi-core parallelization on each node of the database cluster by using adjusted versions of the algorithms from [5] implemented using highly efficient thread-parallelization libraries and inter-node parallelization with well studied group communication operations implemented in libraries following the open standard MPI.

## 1.2. Outline

We start this thesis with an overview of work on topics related to this thesis: We give a short introduction on main-memory and columns-based databases and on the group communication operations used for our algorithms, followed by an overview of research on distributed databases.
In Section 3 we give an insight on the index structures we use for faster access on tuples in the database, the Message Passing Interface (MPI) and the communication Model we use for our analysis. Sections 4 and 5 are the main part of this work. In Section 4 we develop algorithms for the communication efficient execution of queries following certain patterns that appear in many real-world queries and compare these algorithms to other solutions. We then explain how to apply those techniques on chosen queries of the TPC-H benchmark in section 5 and explain our decision for choosing our algorithms over others. In Section 6 we give details on our implementation and explain the limitations of our prototype. Section 7 then describes our evaluation techniques and provides an a posteriori analysis of the scaling behavior in different scaling experiments of the implemented queries. Finally, in Section 8 we review the work of the entire thesis and draw conclusions from our results and describe their applicability in productive systems.

# 2. Related Work

## 2.1. Main Memory Column Store Databases

Lots of work (e.g. [17, 23]) has shown that a column based representation of the data held in main memory works best for OLAP databases and saves memory due to better possibilities for data compression. Also, because of the growing size of main memory on modern machines, data can be hold completely in main memory speeding up query execution time remarkably, while the hard disk is only used for persistent backups [10]. These results lead to many proprietary databases like SAP HANA [8] or EXASOL EXASolution [7] and research databases like HyPer [15] using both at least for parts of their implementation.

While this work mainly focuses on the communication efficiency in query execution, our prototype is implemented as a main memory column store database to achieve fast execution times.

## 2.2. Group Communication Primitives

While we only study the communication efficiency in OLAP databases, there is also some more fundamental research on this topic: Fraigniaud and Lazard [9] study the lower bounds on many group communication operations in networks on different communication models. Bruck et al. [3] analyze the lower bounds on different types of group communications and provide optimal algorithms for those. The results of these two research publications are used for our theoretical analysis and give a deeper insight on the underlying algorithms used in this work.
Sanders and Träff [21] propose an algorithm for (personalized) alltoall communication where every processor on a node has an individual message to send. Their algorithm is used in our prototype for all larger alltoall communications as it behaves better in our experiments than other implementations of the alltoall operation.

## 2.3. Distributed Databases

There has been a large variety of research on the field of distributed databases. Most solutions, however, don't integrate cluster parallelism deep into their implementation but, for example, run a separate database on every node of the cluster and use a middleware to coordinate the cooperation of the nodes. Also, many distributed databases rely on heavy replication of the data among all nodes. While this prevents problems with communication times as all data is available on every node, it prevents the system from truly scaling out as every node still needs enough memory to store all data held in the database. Weidner [26] gives a more detailed explanation of these systems and provides several examples.
Rödiger et al. [19] propose an algorithm for optimal partitioning and partition assignment for distributed joins, however due to the structure of the synthetic data of the TPC-H database these problems can be solved in a trivial way in our case because at least one of the relations taking part in a join is always partitioned by the join-key. They also use the algorithm for solving open shop from [12] to schedule their communication but this only gives high benefits on network utilization for value skewed data while the improvement over round robin for uniform data seems

to be negligible.
Dees and Sanders [5] use full many-core parallelization to execute TPC-H queries but they don't cover inter node parallelization, however adapted versions of their algorithms are used for intra node parallelization in our implementation.
Bernstein and Chiu [1] describe in their work how to reduce the effort required for later joins by doing early semi-join reductions and determine which queries can be solved using semi-join reductions. The approach of semi-joining early is also used by Weidner [26] and we show that it is more beneficial in terms of communication time to do joins at a later stage for many queries. Chen an Yu [4] present how to combine joins and semi joins to get a benefit even for cases were a single semi-join is not profitable on its own.
Koutris [16] proposes the use of Bloom-filters [2] for semi-join reductions to reduce communication time. However, their algorithms don't seem to scale with the number of nodes in a database cluster and there are more space efficient versions of bloom filters: Putze et al. [18] improve space-efficiency for Bloom-filters by using only one hash value for every element inserted into a large hashed bitmap and compressing it using Golomb coding [11]. Sanders et al. [20] also propose a distributed version of these singe-shot Bloom-filters which seem to be an appropriate way for saving communication for filters that can't be evaluated locally if the filter is not queried too often.
Weidner [26] shows that there is a lot of room for improvement on current solutions at least for manually written translation from SQL to C-code but his solutions for some queries still don't seem to scale well. We continue their work and find improvements for their approaches by using algorithms with less communication effort at the cost of more computational work.

# 3. Preliminaries

## 3.1. Data distribution and indices

We assume a distributed database where all tables are distributed horizontally among all nodes, meaning that every node holds some rows for every table and that rows of a table are kept together and not split among multiple tables. We also waive replication of data in most cases. These assumptions are crucial for a system designed to scale out on a theoretically unlimited number of processing nodes.
As it is important for many queries to determine the location of tuples referenced by other tuples on foreign key relations, we construct indices to achieve this in constant time.
For the algorithms presented in this work we mostly use *range indices*. A range index is used for one-to-many relations where the tuples on the *many*-side of the relation are grouped by this foreign key: The index has an entry for every tuple on the *one*-side of the relation that points to the first tuple on the *many*-side that references the first tuple. So for the tuple at position $i$ the tuples referencing that tuple can be found in the range index in the interval $[range(i), range(i+1)]$.
Another index that can be useful in a distributed database is one that maps foreign keys to the node where the referenced tuple is located. In our case this can be done easily by storing for every node the key of the first tuple of every table on every node. So if we want to find the node of a tuple with key $x$ we just have to find the node $i$ with $index(i) \leq x < index(i+1)$. This works for all cases where the tables are ordered by their key and the database is distributed by a range partitioning [6].

Using horizontal partitioning of the data leads to two major different situations when performing a join. We adopt the classification from Weidner [26] and denote joins that can't be performed without accessing data on another node as *remote join* or *on a remote join path* and *local joins* respectively. The efficient execution of these *remote joins* is the main topic of this thesis.

## 3.2. Message Passing Interface (MPI)

In order to execute queries in a distributed database, the nodes of the cluster running the database need to communicate at some points of the execution. In the interest of doing so in an efficient way we use sophisticated group communication patterns. A group communication is an operation that many (or in our case all) nodes take part in.
The standard *Message Passing Interface (MPI)* provides a library specification including many of these group communication operations among with other functionalities and has been implemented for several programming languages including C, C++, C# and Java.

The operation we use most in our algorithms is the (personalized) **alltoall** communication where every node holds some data to send to every other node: If the program runs on $P$ nodes, every node holds $P$ blocks of data (possibly differently sized) and sends block $i$ to node $i$.

Other operations we use are reduce, allreduce, gather, allgather, scatter and broadcast:

- **Reduce**: The data from all nodes, each of the same size ($n$), gets combined by an associative operation. The result is gathered at one of the nodes (called the root) and has size $n$.
- **Allreduce**: Same as reduce, but the result lies on all nodes.
- **Gather**: The data from all nodes gets gathered at one root node resulting in one block of data with a size of $\sum_{i=0}^{P-1} size(i)$ where $P$ is the number of nodes taking part in the gather operation and $size(i)$ is the size of the data block on node $i$.
- **Allgather**: Same as gather, but the result lies on all nodes.
- **Scatter**: Opposite of gather. The root node sends one block of data to each node (possibly of different sizes). The $i$th block is sent to node $i$.
- **Broadcast**: The root node sends the same block of data to all other nodes.

Further study on group communication can be found in [9], [21] and [3].

## 3.3. Communication Model

Stonebraker [25] describes three different architectures for high transactionrate multiprocessor systems:

- **Shared Memory**: All processors share one central memory
- **Shared Disk**: Every processor has its own memory but one disk is shared among all processors
- **Shared Nothing**: The processors share no resource. Every processor has its own memory and disk and the processors are linked by a network

By the classification of Stonebraker the architecture we design our algorithms for is *shared-nothing* as we only cover inter-node parallelization and use the algorithms from [5] for intra-node parallelization. However, as in a usual cluster every node has multiple processors sharing the same memory, our implementation supports a combination of shared memory and shared nothing: We support full intra-node parallelization to use all the available processors of a node (shared memory). Between the nodes of the cluster there is no shared resource except for the linking network, that is used to exchange messages between the nodes (shared nothing).

For our analysis we use the one-port fully connected message-passing system where in every communication round every node can send and receive data from one other node and all nodes are equally distant.
To evaluate communication complexity we use the linear model, which states that the time required for sending $n$ units of data is $\alpha + n \cdot \beta$ where $\beta$ is the time for sending one unit of data and $\alpha$ is some overhead yielded by every communication.
These models are also used by Bruck et al. [3] and Fraigniaud et al. [9] in their analysis.

# 4. Distributed Query Execution

## 4.1. Partitioning

When (equi-)joining two tables the optimal partitioning of the data is when both tables are copartitioned after the join key, meaning that all tuples of both tables with the same value on the join key should be on the same partition of the distributed database thus a join on these two tables can be executed locally needing no communication effort.
There are some rules that can be used to determine tables that can be copartitioned by foreign-key relations by possibly giving up the partitioning by the primary key of one of the tables.
Let the partition graph for a database schema be defined as follows:

- Tables in the database are modeled as nodes $v \in V$
- Foreign key relations for tables that are **not** copartitioned by that key are modeled as directed *remote* edges $r \in R$
- Foreign key relations for tables that are copartitioned by that key are modeled as directed *local* edges $l \in L$

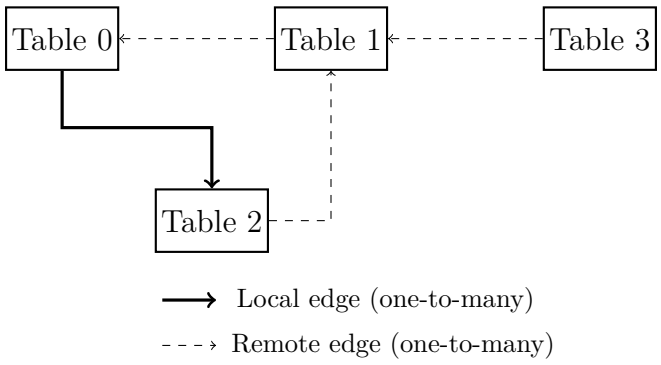


Figure 1: Example of a partition graph

In this graph possible tables to be copartitioned can be found easily. If there is a remote edge $r = (t1, t2) \in R$ such that the graph $(V, \{r\} \cup L)$ is acyclic, then $r$ can be made local by repartitioning $t2$ to the corresponding elements in $t1$ and all $t \in V$ with $(t2, t) \in L^*$ respectively.
Note that in general a table can only be copartitioned with one other table on it's foreign key, so if there are two edges $(t1, t), (t2, t)$ fulfilling the requirements above, $t$ can only be copartitioned with $t1$ or $t2$, not both.

In the example table schema from Figure 1 $Table$ 1 can be repartitioned by its foreign key to $Table$ 3.
Additionally $Table$ 0 can be repartitioned to $Table$ 1 at what $Table$ 2 must be repartitioned to match the new partitioning of $Table$ 0.
After that no other repartitionings are possible in this schema although there is still the edge $r = (Table\ 2, Table\ 1) \in R$ because $(V, \{r\} \cup L)$ has the cycle $(Table\ 0, Table\ 2, Table\ 1)$. The resulting graph can be seen in Figure 2.

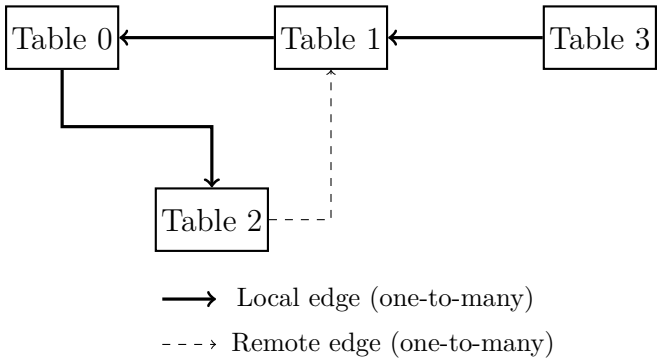


Figure 2: Example of a partition graph after copartitioning the tables

In this example it is easy to see that there can be more than one partition graph for a given table schema where no more repartitionings are possible.

While having many co-partitioned tables in the database will increase join performance by avoiding unnecessary communication, it might be more beneficial to keep some tables partitioned by their primary key in order to allow fast access on successive tuples. There has to be made a trade-off between these two arguments for the specific tables in a database.

For an example were we use the possibility to repartition tables in this way see Section 5.2.1.

## 4.2. Global result reduction for top k selection queries

An important factor in designing parallel algorithms is avoiding bottlenecks which is achieved by algorithms with an optimal load balance.
In order to accomplish that target all of our top $k$ selection algorithms construct a local result on every node, aiming not to result in a bottleneck, and then derive the global result for the query via a final global reduce operation: Every node selects the first $k$ elements from its partition of the tuples in the result set - then the global top $k$ tuples get derived from the local top $k$ results by a global reduce operation. Figure 3 illustrates this technique for a possible implementation of a reduce operation as a binary tree.

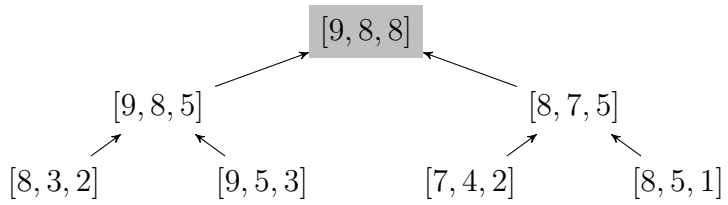


Figure 3: Local result to global result reduction using a binary tree. The global top 3 results get derived from the local top 3 results on 4 nodes using a reduce operation.

Let $P$ be the number of nodes, *size* the amount of data to transfer for every tuple and $m$ the number of tuples in the result set before eliminating all of them except the first $k$. Then, using the lower bounds proven in [9], the total costs for deriving the global result from local results using the described reduce function are

$\log_2(P) \cdot k \cdot size$ compared to $(P-1) \cdot k \cdot size$ for gathering all local results on one node and selecting the first $k$ tuples from it, or even $(P-1) \cdot m \cdot size$ for gathering the entire result set on one node and then eliminating all tuples but the top $k$.

This technique was already used in [26] and we use it for all of our implemented queries (Sections 5.1, 5.2 and 5.3).

## 4.3. Late joins

An approach often used to reduce computation time is to do a semi-join reduction [1] early and only materialize the results yielding from the calculations on the reduced tuples. To do this for remote join paths in a distributed setting for example a full bitset [26] or a bloom filter [16] for all tuples fulfilling the filter condition is replicated among all nodes (possibly compressed). This approach aims to reduce local calculation time. However, Rödiger et al. [19] mentioned that CPU speed grew way faster than network speed over the last decades and predicts this to continue in the near future, thus saving communication time is going to be more important than saving CPU clock cycles. To reduce communication volume for queries with a locally evaluable and a remote filter condition, the local result can be partly calculated without joining the tables and after that a bitset for the filter on the remaining keys lying on the remote join path can be requested explicitly if the remote location of the corresponding tuples is known. While this technique can often save high degrees of communication it does cost more computation time and storage on each node because tuples that get filtered out by the remote filter condition have to be processed and stored first.

Figure 4 illustrates the technique using an example query on an example database: There is a filter condition on the remote join path between $l$ and $r$ that has to be evaluated efficiently. First the values get aggregated by the join key and filtered locally. Then they are joined on the remote path which acts as a filter. No node has to communicate all of its information to every other node, e.g. node $j$ does not need any information about the remote filter on join keys $w$ and $z$ because it has filtered out $z$ locally.
It is not shown in the illustration how the join would be processed: The nodes on the left would send the join keys for which they hold results to the nodes on the right. The nodes on the right would return a bitset containing information about which of the requested keys fulfill the filter condition.

We analyze this approach in comparison to distributing a bitset for the remote filter over all nodes. We don't compare our solution to the version using Bloom filters because Bloom filters are only profitable for very small selectivities.

**Theorem 4.1.** *Requesting the required keys for a remote join and answering a bitset is more efficient regarding communication complexity if the inequation $f < \frac{1}{8k+1}$ holds where $f$ is the selectivity on the locally evaluable filters and $k$ is the amount of bytes that have to be send for every key in order to request the remote filter on it.*

*Proof.* For the full bitset distributed over all $P$ nodes, one *allgather* operation is needed with $\frac{n}{8 \cdot P}$ bytes of data to be sent from every node where $n$ is the size of the table used for the filter on the remote join path (assuming the locations of every tuple

in the bitset can be determined in some way on every node). According to Bruck et al. [3] the lower bound on communication complexity for this is $C_{full} = \frac{n}{8 \cdot P} \cdot (P-1)$ bytes.

Our approach needs two phases: An *alltoall* operation to request the required join keys and another *alltoall* operation to return a bitset for the filter on the requested keys.
Because the table was already filtered locally thus not every tuple has to be filtered on the remote join path, each node only has $\frac{n \cdot f \cdot k}{P}$ bytes of data to sent. Again, according to [3] the lower bound on communication complexity for this is $C_{lateRequest} = \frac{n \cdot f \cdot k}{P} \cdot (P-1)$ bytes. For the answered bitset this yields $C_{lateAnswer} = \frac{n \cdot f}{8 \cdot P} \cdot (P-1)$ bytes

In order to find the point where our approach performs better than the variant of

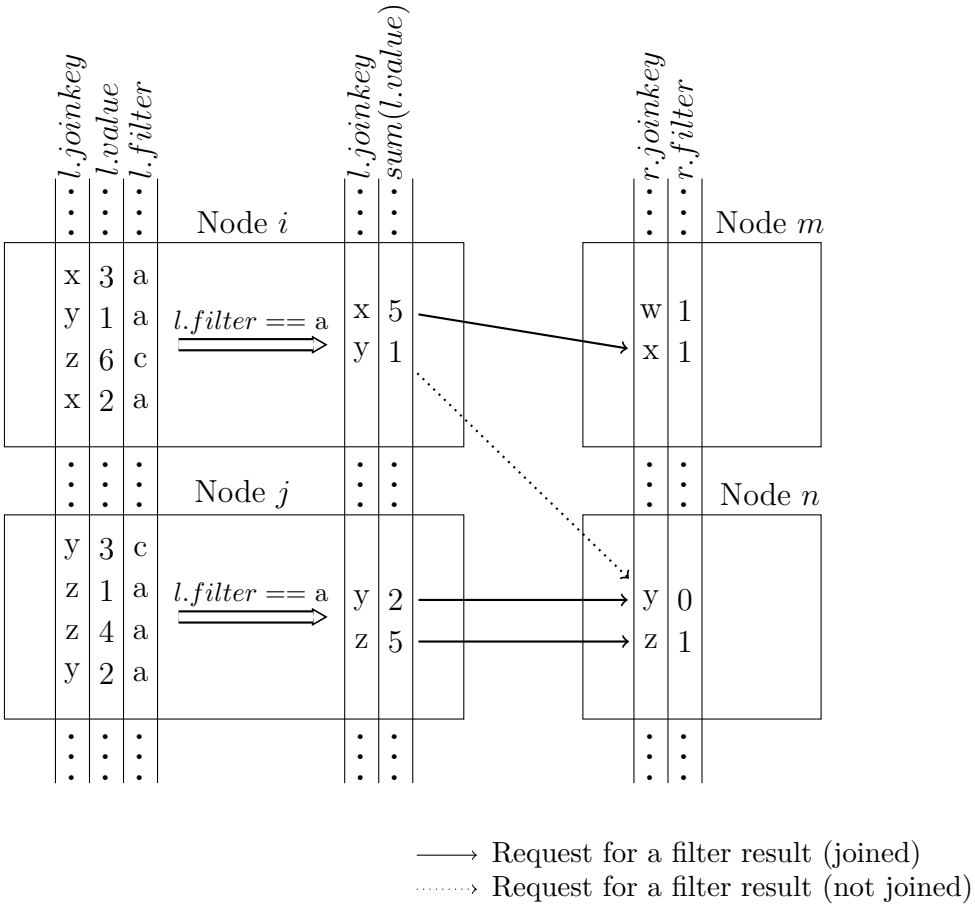


Figure 4: Example of a late join for the query
`SELECT l.joinkey, sum(l.value) FROM l, r WHERE l.joinkey = r.joinkey AND l.filter = a AND r.filter = 1 GROUP BY l.joinkey`.
$l$ is not partitioned after *joinkey* but $r$ is, thus the join between $l$ and $r$ is on a remote path.
First the values get aggregated by the join key and filtered locally. Then the filter on the remote join path is evaluated.

replicating a full bitset we evaluate the inequation $C_{lateRequest}+C_{lateAnswer}-C_{full} < 0$:

$$\begin{aligned} & (\tfrac{n \cdot f \cdot k}{P} + \tfrac{n \cdot f}{8 \cdot P} - \tfrac{n}{8 \cdot P}) \cdot (P-1) & \\ = \quad & \frac{n \cdot f \cdot (8k+1) - n}{8P} \cdot (P-1) & < 0 \\ \overset{P \geq 1}{\Rightarrow} \quad & n \cdot f \cdot (8k+1) & < n \\ \overset{n \geq 0}{\Rightarrow} \quad & f \cdot (8k+1) & < 1 \\ \Rightarrow \quad & f & < \frac{1}{8k+1} \end{aligned}$$

Figure 5: Proof for the break even point of early and late joins.

□

We ignored latencies and startup times for this comparison due to very large data sizes, Bruck et al. [3], however, showed that the lower bound on the number of communication rounds for *alltoall* operations is way higher when reaching the lower bound on the data transferred in a sequence ($P-1$ in comparison to the overall lower bound of $\log_2 P$).

Note that in many cases most of the communicated information can be compressed, e.g. by using differential- or run-length encoding in combination with Golomb coding [11].

An example for the application of this technique can be found in Sections 5.1 and 5.3.

### 4.3.1. Exploiting previously communicated information

In some query execution plans we can save communication volume by exploiting previously communicated information. An example for this are special cases of query plans where a semi-join as described in Section 4.3 is done:
When Node $A$ requests a remote filter on node $B$, it has to send the join keys to node $B$. After receiving the result and computing the next intermediate result using that information on node $A$, we might need to send more information to node $B$ which belong to some or all of the join keys transferred earlier. Here we can skip sending the join keys again by sending the additional information in a way that allows us to match the new data with the previously sent join keys e.g. by using the same order.

We exploit previously communicated information in Section 5.1 to determine tuples in the result set and in Section 5.3 to send additional data, where we provide additional explanations for this technique relevant to the specific examples.

### 4.3.2. Lazy top-k filtering

A common pattern in decision support queries is to aggregate values by a key and return only the top $k$ results filtered by a join attribute on the aggregated values. A special case of this problem in a distributed Database is when the tables to be aggregated are partitioned by the aggregation key but not by the join key.
This thesis provides an efficient solution for this problem by evaluating the remote filter lazily thus only needing to get the remote filter results for a number of Elements

expectable linear in $k$.
First the aggregated values for the whole table grouped by the aggregation key have to be calculated and sorted locally needing no communication. Second, the remote filter has to be evaluated for single keys in the sorted aggregate result set until the filtered top $k$ elements are determined. Last, the local top $k$ results have to be reduced to get the global top $k$ elements (see Section 4.2). As this yields a communication volume independent from the number of tuples in the database, we expect this technique to scale fairly well. Figure 6 illustrates the algorithm.

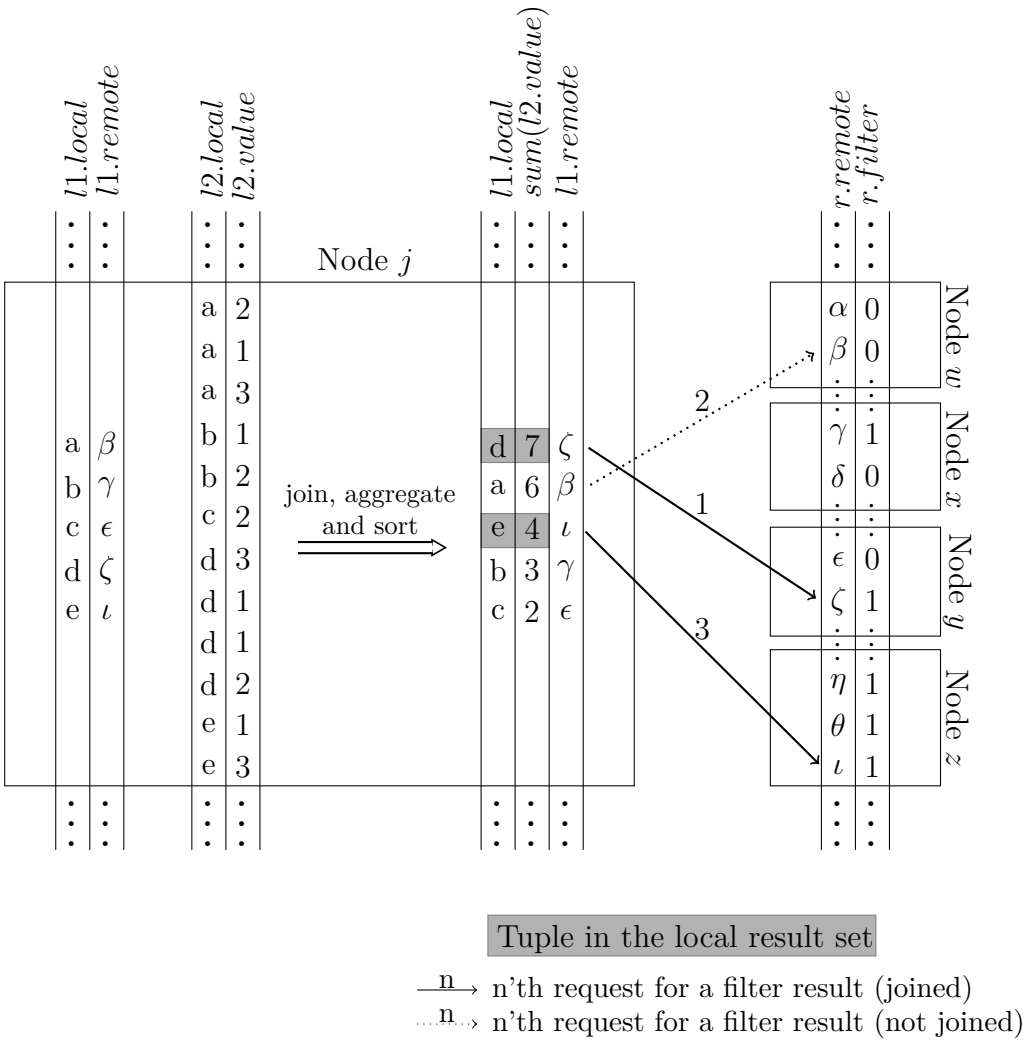


Figure 6: Example of a lazy filter on the top 2 results of the query
`SELECT l1.local, sum(l2.value) FROM l1, l2, r WHERE l1.local = l2.local AND l1.remote = r.remote AND r.filter = 1 GROUP BY l1.local ORDER BY sum(l2.value)`
First the values are aggregated locally and sorted by their sum. To determine the local top 2 results, the filter results on the remote path are requested one after another until the top 2 sums matching the filter condition are found.

**Theorem 4.2.** *Filtering the aggregates until the top $k$ elements are found needs expected $\frac{k}{p}$ remote accesses on each node where $p$ is the probability of getting a positive filter result.*

*Proof.* We consider the application of a filter to be a Bernoulli experiment where a match on the filter condition is a success. We denote the probability for a success with $p$. As we need to evaluate a sequence of Bernoulli experiments we get a binomial distribution.
We have to find the number of trials $n$ needed to have at least $k$ successes.
The expectation of a binomial distribution is $n \cdot p$, thus we have to solve the inequation $n \cdot p \geq k \Rightarrow n \geq \frac{k}{p}$. □

As every communication yields some startup overhead, a filter on more than one key should be requested at a time. By starting with requesting the filter on 2 keys and doubling that value until the top $k$ results are determined, the number of keys requested is at most double of the minimum amount (the worst case would be if the minimum amount of requested keys is $2^i + 1$ for some $i$ where this method would result in requesting $2^{i+1}$ keys) while saving most of the startup overhead. As the minimum number of tuples to filter is $k$, this can also be used as a starting value.

For a sample application of this algorithm see Section 5.2.2

# 5. Application in TPC-H

The TPC-H benchmark [24] is an OLAP benchmark provided by the Transaction Processing Performance Council (TPC). It provides a Database schema, a data generation tool, 22 decision support queries and 2 refresh functions. The amount of data generated can be modified by a scale factor $SF$, specifying the size of the uncompressed data in GB. A graphical representation of the schema can be found in Figure 7.

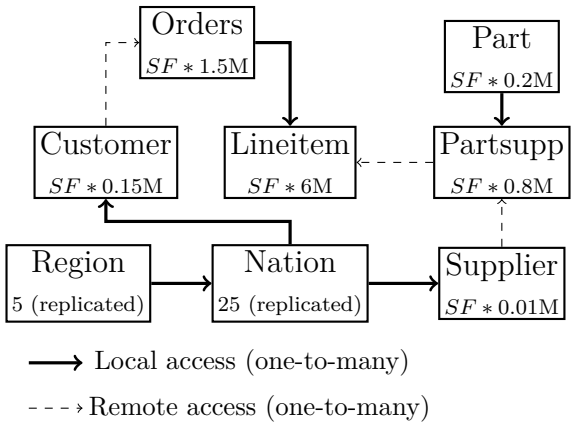


Figure 7: Partition graph (see Section 4.1) of the TPC-H tables. All tables are presented with their name, size in million and if the size depends on the scale factor $SF$. Transitive relations are not illustrated. *Source: [26]*

We replicate the *Region* and *Nation* tables over all nodes because of their small size (5 and 25 tuples) and the frequent access on them in the benchmark queries. Furthermore, for Query 3 (Section 5.2), we make an adjustment to the partition graph. We do not do this for any other query because it is not needed, however all other algorithms would still work with the changed partitioning of the tables.

## 5.1. Query 2

Query 2 of the TPC-H benchmark finds for each *part* of given size and type the *supplier* from a given *region* with the lowest price for that part and returns the top 100 results ordered by the suppliers account balances.

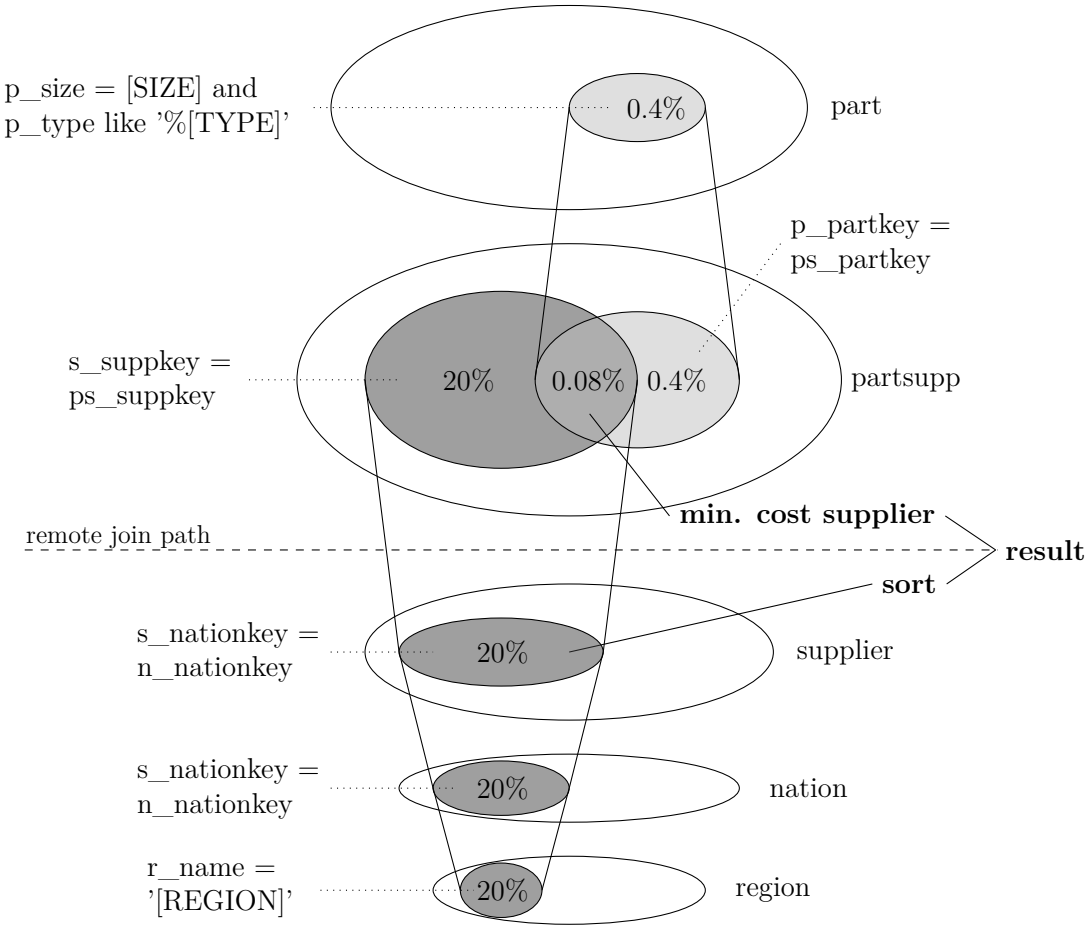


Figure 8: Joins in Query 2. Tables are illustrated as sets and joins as projections from one subset into another. Percentages in subsets are ratios in comparison to the outermost set. *Visualized as presented in* [13]

For this query there is a remote join path between *partsupp* and *supp* (see Figure 8) where a communication efficient algorithm is to be used.

Our implementation for Query 2 saves communication volume by joining the two tables as late as possible, thus not needing to join every tuple in the two tables. It also utilizes that some information transfered for a semi join can be reused to reduce communication volume in a later stage of the execution (line (iv)).

(i) Filter all parts by their size and type and find the corresponding suppliers from the *partsupp* table

(ii) Send the left supplierkeys (0.4% of the *partsupp* table) to their corresponding nodes, filter them by their region and return a bitset for the filter result to the requesting node

(iii) For each left part find the minimum cost suppliers from the remaining suppliers using an adapted version of the algorithm from [5]

(iv) Send a bitset, specifying for each positively filtered supplier whether it is in the result set or not, to the corresponding nodes

(v) Compute a local result and derive a global result using a global reduce operation

Doing a late join here is more efficient than distributing a bitset for the entire *supplier* table because it can be derived from the specification of the TPC-H Benchmark [24] that the local filter on *p_size* and *p_type* has a selectivity of $\frac{1}{250}$, thus the amount of data that is communicated for requesting the remote filter on each supplier can be up to 31 bytes without having a larger communication volume than for the full bitset (see Section 4.3) and the supplier key is only 4 bytes.
As steps (ii) and (iv) are highly dependent on the size of the tables, we expect this algorithm to require more time when adding more nodes but keeping the amount of data on each node.

## 5.2. Query 3

Query 3 retrieves the top 10 unshipped *orders* at a given date with the highest revenues filtered by the customers market segment.
Figure 9 illustrates the joins and filters done by Query 3 for the default data partitioning of the TPC-H tables.
We implement Query 3 in two different ways. By using another partitioning of some tables and by evaluating the remote filter lazily.

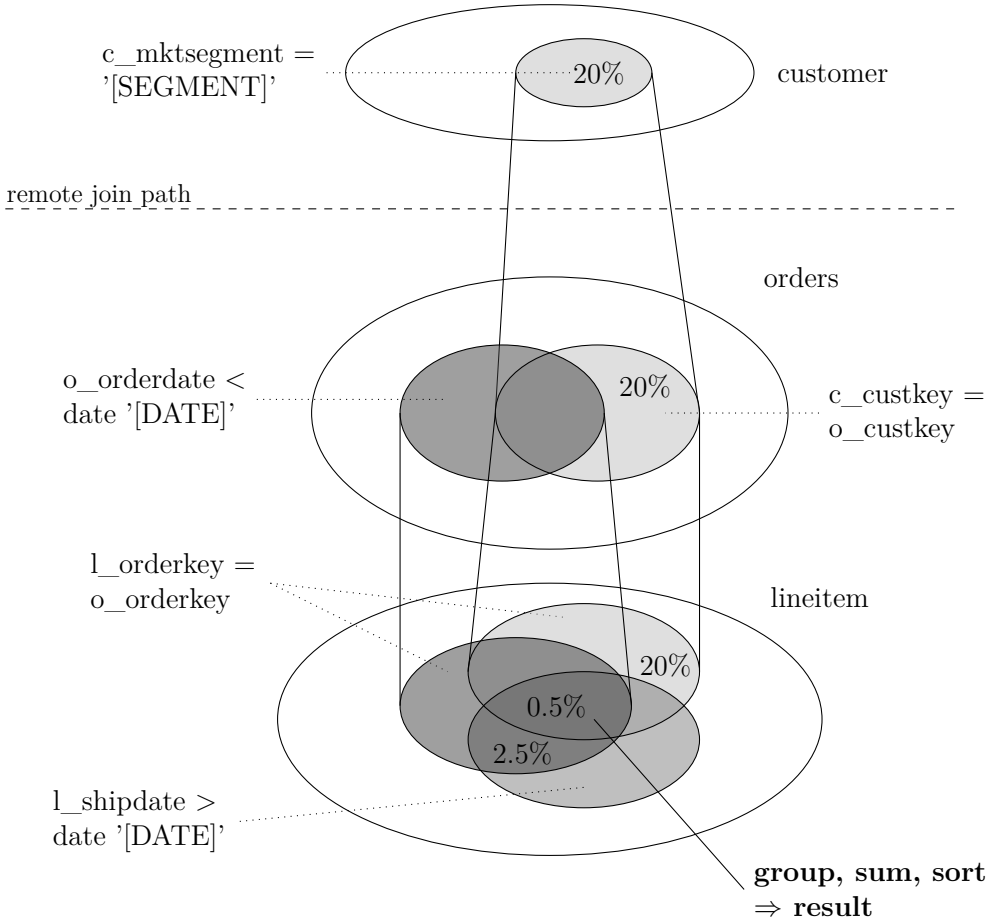


Figure 9: Joins in Query 3. Tables are illustrated as sets and joins as projections from one subset into another. Percentages in subsets are ratios in comparison to the outermost set. *Visualized as presented in* [13]

### 5.2.1. Repartitioning

One variant of implementing Query 3 is by repartitioning the *orders* and *lineitems* table to be copartitioned with their corresponding customers. This does not remove any copartitioning of the other tables, hence it is possible to execute all other queries in the same way as they would be executed without the repartitioning. Also sequential access on subsequent orders is never needed in the TPC-H Benchmark.
After repartitioning the tables, the ten orders with the highest revenue can be determined locally using the algorithm from [5]. Finally the local results have to used to find the global top ten orders using a global reduce function. Using this algorithm the only communication needed is the communication effort for the reduction to a global result.

### 5.2.2. Lazy Evaluation

The second variant for Query 3 uses the lazy top-k filtering algorithm described in Section 4.3.2 so the query consists of three sub-queries:

(i) Calculate the revenues of all *orders* filtered by date locally and sort the result by revenue.

(ii) Filter the top orders until the top ten local results are determined

(iii) Use a global reduce operation to derive the global top ten orders from the local top ten orders on every node

Using this algorithm every PE only has to communicate to get the filters on expected 50 orders plus the communication needed to reduce the local results to a global result, i.e. the communication effort is independent from the database size. However, it is not independent from the number of nodes used as the number of filter results every node has to retrieve does not change (but is fairly low).

## 5.3. Query 21

Query 21 identifies the *suppliers* form a certain *nation* who have failed to ship parts of an *order* on time where all other *suppliers* shipped on time and returns the top 100 *suppliers* with the highest number of delayed shipments. We illustrate the joins and filters necessary for this query in Figure 10.

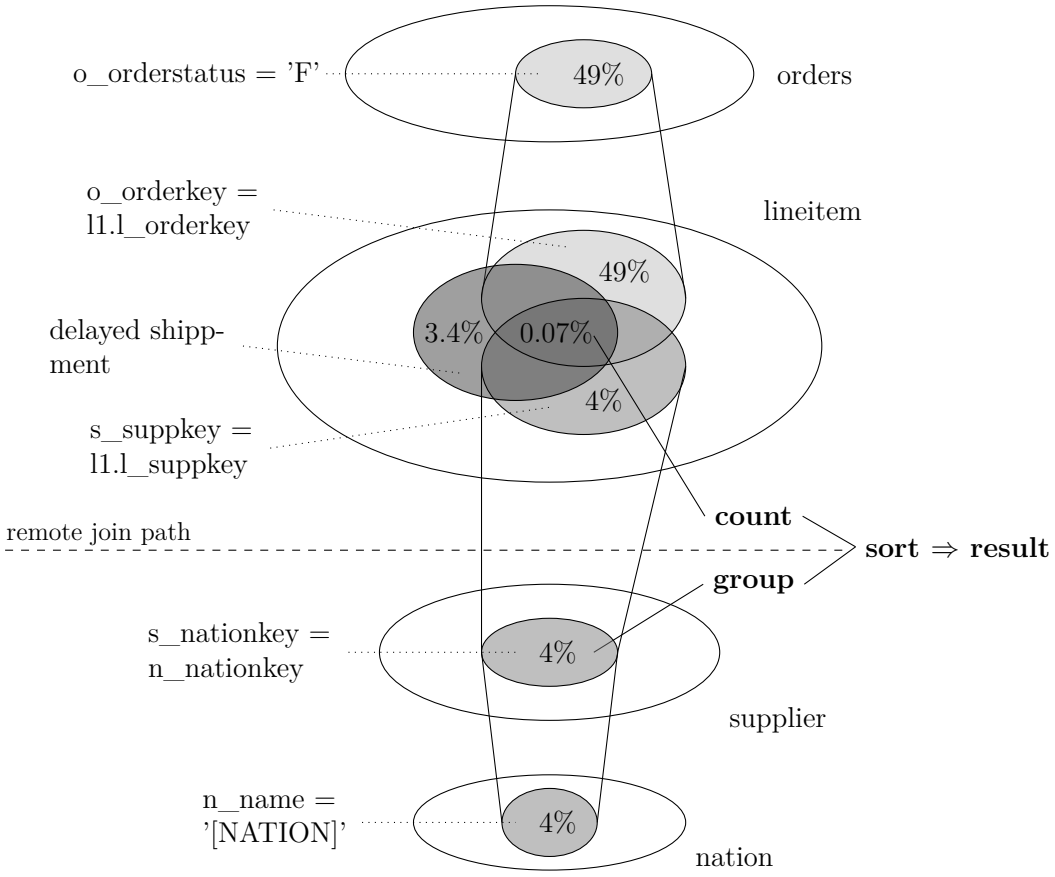


Figure 10: Joins in Query 21. Tables are illustrated as sets and joins as projections from one subset into another. Percentages in subsets are ratios in comparison to the outermost set. *Visualized as presented in* [13]

In this query the remote join path between *lineitems* and *suppliers* causes a high communication volume, however we can still save some communication overhead over naïve solutions. Because in a large cluster with many nodes not every node holds information for every *supplier* we can request the required *suppliers* explicitly thus only have to communicate a bitset for the *suppliers* required on a node and not for the entire *suppliers* table.
Using this technique our algorithm for Query 21 works like this:

(i) Find the number of delayed shipments for each *supplier* locally using the algorithm from [5] without the filter on the suppliers *nation*

(ii) Request the filter information for the *nation* key of all suppliers with at least one delayed shipment from the corresponding nodes

(iii) Return an individual bitset for the requested *suppliers* to each node

(iv) Send the locally computed number of delayed shipments for suppliers with a positive filter result to the node holding the corresponding supplier information

(v) Determine the local top 100 results with the fully aggregated number of delayed shipments for each supplier

(vi) Reduce the local top 100 to the global top 100 results and completely materialize the result by requesting the names for the *suppliers* in the result set

Note that when sending the partial aggregated number of delayed deliveries in line (iv) it is not necessary to send the corresponding supplier keys as the receiving node has the information about which suppliers are going to be sent from lines (ii) and (iii). Therefore it is important to send the information in lines (ii) and (iv) in the same order so the receiving node can match pairs of supplier keys and values belonging together.

To show that evaluating the remote filter condition at this later stage instead of as the first step (like Weidner [26] did in his work) we use the inequation from Section 4.3:
We found the selectivity of the filter conditions that can be evaluated locally empirically. In Figure 10 we can see that see that 3.4% of the orders have a delayed shipment by a singe supplier and 49% of the orders remain after filtering by the orderstatus. As these two filters are uncorrelated (which we validated by experiments), the selectivity of the filters that get evaluated locally is $3.4\% \cdot 49\% \approx 1.7\%$. By solving the inequation we find that the amount of data communicated for requesting the filter on each supplier can be up to 7 bytes and even sending the entire supplier key only takes 4 bytes.

To achieve a communication volume closer to the theoretical minimum both the requests for filter information and the answered bitsets might be compressed using Golomb coding [11] in combination with run-lenght- and differential coding.

Just like in Section 5.1, the communication time of this algorithm is highly dependent on the input size. Also, due to the implementation for step (i) we adopted from Weidner [26], the local computation time is dependent on the overall table sizes too even if the amount of data each node holds doesn't change.

# 6. Implementation Details

We implement our algorithms for Queries 2,3 and 21 of the TPC-H benchmark in a prototype of a distributed main memory column-stored database. We extend the C++ implementation from Weidner [26] with our algorithms, hence we also use the MPI implementation OpenMPI for inter node parallelism and Intel TBB for intra node parallelization. In some cases, i.e. parallelization of simple loops we use OpenMP for thread parallelism. For efficient communication via MPI we use custom data types and reduction functions to avoid unnecessary startup overhead for multiple communication rounds. Also, for large alltoall communications we exchange the OpenMPI implementation with the algorithm from Sanders and Träff [21] which, in most cases, performs better than the standard implementation for large data sizes in our experiments.
We also use Weidners extended data generator for in-memory output of the generated data.
For compression we use our own implementation of Golomb coding [11] using boosts *dynamic_bitset* which might lack optimal performance, thus by using more performant implementations or other compression algorithms, runtimes might improve by some degree.
For thread parallel construction of messages to send, we construct each message on a different thread. This results in adequate runtimes for cases where there are more messages to construct than hardware threads available. However, if that is not the case, i.e. small numbers of nodes, there will be idle threads resulting in suboptimal performance of our implementation.

## 6.1. Limitations

As we extended the implementation developed by Weidner [26], our prototype still does not fulfill the requirements of a full database management system (DBMS). The limitations are the same as in their work (directly adopted):

- No transaction and session handling
- No updates and inserts
- No fault tolerance
- No execution of arbitrary SQL

In particular we translated the queries into C-Code manually, so our implementation consists of hard-coded execution plans for all three queries.

# 7. Evaluation

## 7.1. Experimental environment

We evaluate our algorithms on the *Institutscluser II* of the *Karlsruhe Institute of Technology* at the *Steinbruch Centre for Computing*. The cluster consists of 400 nodes with two E5-2670 Intel Xeon octa-cores with 2.6 GHz, $8 \cdot 256$ KB L2 cache, 20 MB L3 cache and 64 GB of main memory each, connected by an InfiniBand 4X QDR. For our experiments we use up to 128 nodes as that is the maximum number of nodes a user can use for a job on the cluster. In a micro benchmark we achieved a runtime of 0.0405 seconds for an alltoall communication (which is the most used communication operation in our algorithms) on 128 nodes with 50 MB of data on each node.
We compiled our implementation using GCC 4.8.2 with optimizations and used version 1.6.5 of the OpenMPI library.

## 7.2. Weak scaling experiments

We evaluate our algorithms with two different types of scaling experiments, the first being the weak scaling experiments, where we increase the number of nodes and the size of the database simultaneously, i.e. linear. This kind of experiment lets us evaluate the growth of parallel overhead dependent on the size of the problem. This gives us an insight on the behavior of the system when scaling it out on more machines in order to handle larger problems, which is the easiest solution to react to growing database sizes in a real world scenario.

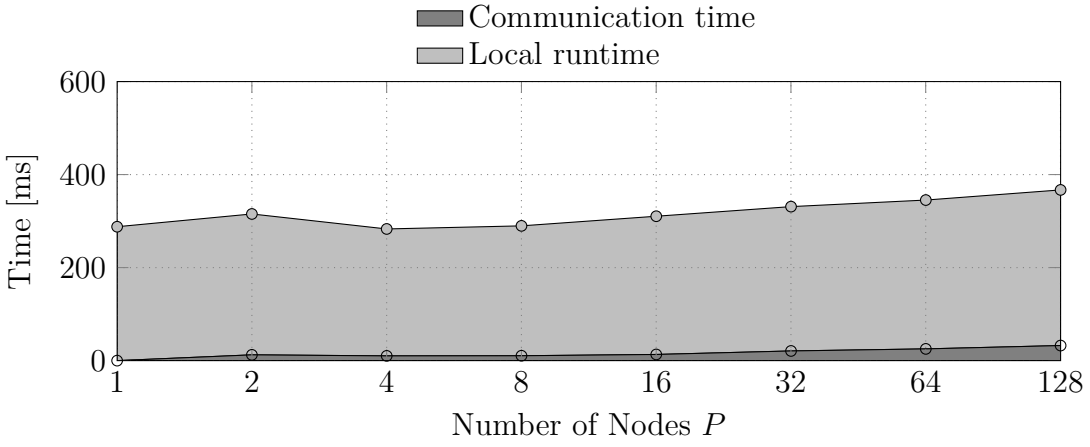


Figure 11: Query 2 weak scaling experiment. Runtime and communication time for different numbers of nodes $P$ and $SF = 1000 \cdot P$

In Figure 11 we see that our implementation for Query 2 scales well in a weak scaling experiment. The local part of the execution is independent from the local database size. Only the remote join path and the global result reduction causes some dependency from the number of nodes and the database size: By adding more nodes and

increasing the table sizes, the filter results on more suppliers have to be requested over the network, which is why we see a slight increase in runtime for increasing numbers of nodes. This additional time needed is mostly caused by communication to perform the remote join between *supplier* and *partsupp*, leading to larger communication volume and also more communication rounds of the Alltoall-algorithm (from Sanders and Träff) which results in more communication startup overhead. Additionally for more nodes, some more work has to be done for construction of the the messages and preparation of the communication.

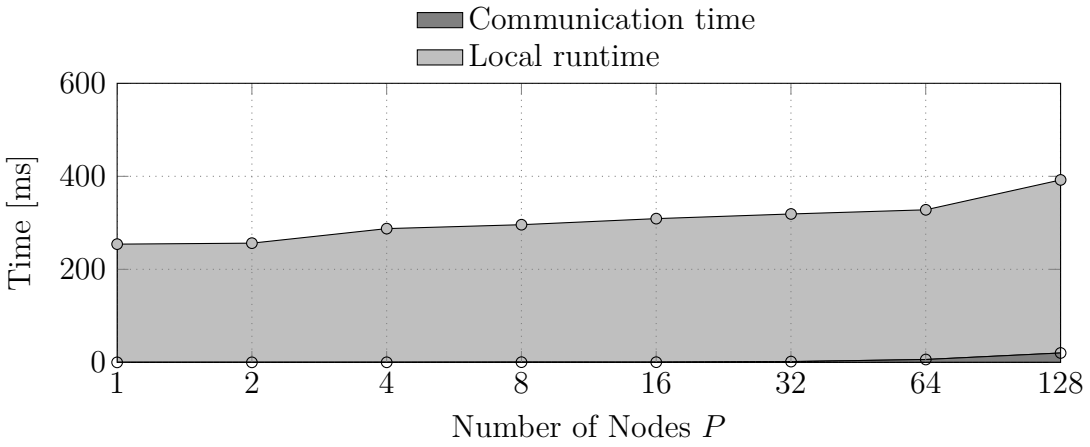


Figure 12: Query 3 with lazy filter evaluation weak scaling experiment. Runtime for different numbers of nodes $P$ and $SF = 1000 \cdot P$

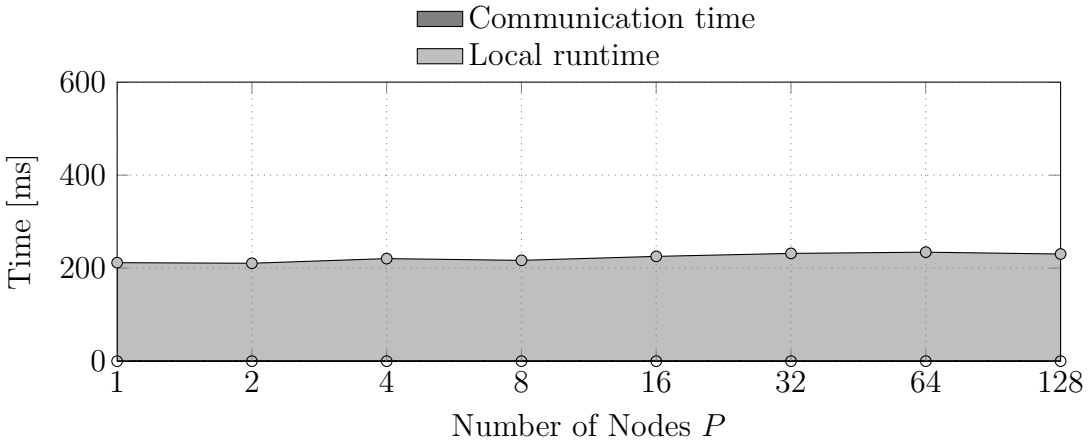


Figure 13: Query 3 with repartitioned tables weak scaling experiment. Runtime for different numbers of nodes $P$ and $SF = 1000 \cdot P$

Figures 12 and 13 present our weak scaling results for Query 3.
The version with the optimized partitioning of the tables has - as expected - almost constant runtime with increasing number of nodes, as the only communication needed is for the reduction used to determine the global result based on local results. Also the local calculations are completely independent from the overall table sizes and no time is used to construct messages or determining which nodes a tuple lies on because it is known that all tuples needed are located on the same node.
For the version with the lazy filter evaluation, we also see pretty good scaling behavior, there is an increase in runtime for large numbers of nodes, which is caused

by the communication needed to evaluate the remote filter. As the number of nodes increases, there are more nodes to request filter results from resulting in smaller, but more messages to construct and send. Furthermore every additional node also has to request for filter results, so both the overall communicated data and the number of communication rounds increase in the weak scaling experiments. However, as every node only has to request the filter results on a small (almost) constant amount of orders, communication doesn't become a large proportion of the query runtime.

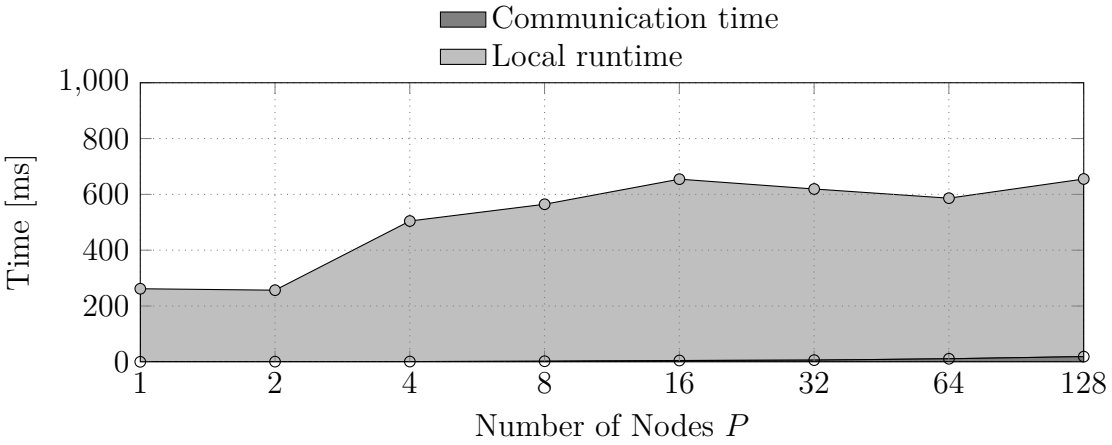


Figure 14: Query 21 with compression weak scaling experiment. Runtime and communication time for different numbers of nodes $P$ and $SF = 1000 \cdot P$

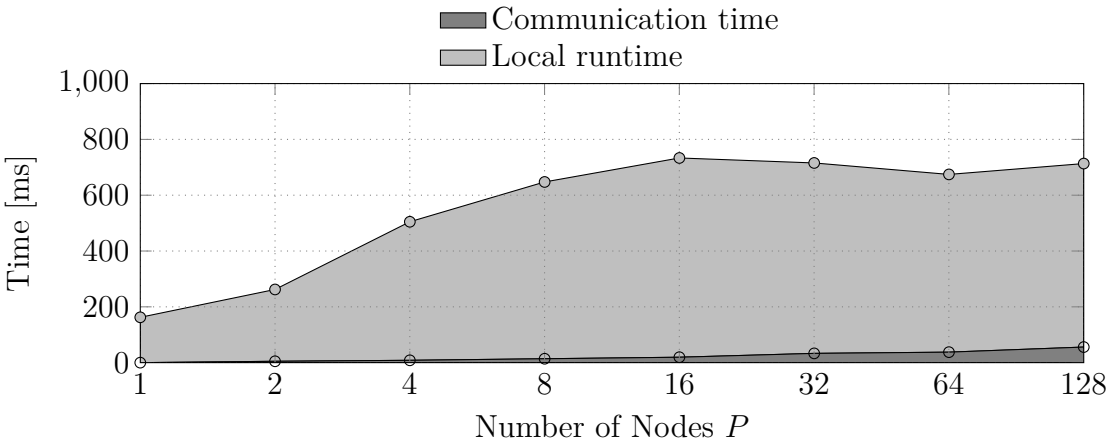


Figure 15: Query 21 without compression weak scaling experiment. Runtime and communication time for different numbers of nodes $P$ and $SF = 1000 \cdot P$

In Figures 14 and 15 we see that Query 21 does not scale as well as the other queries. This is mainly caused by the implementation of the first step of our algorithm that we adopted from Weidners implementation: Like Weidner, we construct an array with one entry per *supplier* in the database on every node. In this array we store the number of delayed shipments for every supplier. As the array gets larger with larger scale factors, cache-efficiency decreases drastically, resulting in higher execution times. Also, the construction of the messages for alltoall communication becomes more expensive for larger numbers of nodes.
For higher numbers of nodes, the runtime does not increase that fast any more. We assume the reason for this to be that not every node needs to access every memory

page of the large array, resulting in the cache efficiency not getting worse anymore. For scale factor 12 800 on 128 nodes, about 10% of the runtime is used for compression, so by using more performant compression algorithms, the runtime could be decreased by a considerable degree.

Table 1: Runtimes and communication times in ms for weak scaling experiments. $SF = 100 \cdot Nodes$ for Query 3 and Query 21 and $SF = 1000 \cdot Nodes$ for Query 2.

| Nodes | Query 2 | | Query 3 (lazy) | | Query 21 (compr.) | |
|---|---|---|---|---|---|---|
| | Overall runtime | Comm. time | Overall runtime | Comm. time | Overall runtime | Comm. time |
| 1 | 287.90 | – | 254.19 | – | 261.84 | – |
| 2 | 315.35 | 12.43 | 256.30 | 0.04 | 256.70 | 0.62 |
| 4 | 283.13 | 10.20 | 287.56 | 0.13 | 504.53 | 0.91 |
| 8 | 289.77 | 10.49 | 295.98 | 0.40 | 564.58 | 2.08 |
| 16 | 310.45 | 13.25 | 309.04 | 0.36 | 654.53 | 4.64 |
| 32 | 331.14 | 20.80 | 319.05 | 1.67 | 619.30 | 6.03 |
| 64 | 345.26 | 25.23 | 328.01 | 6.14 | 586.40 | 10.93 |
| 128 | 367.05 | 32.41 | 392.25 | 20.15 | 655.15 | 18.63 |

| Nodes | Query 3 (repart.) | | Query 21 (uncompr.) | |
|---|---|---|---|---|
| | Overall runtime | Comm. time | Overall runtime | Comm. time |
| 1 | 211.77 | – | 162.44 | – |
| 2 | 210.54 | 0.01 | 262.26 | 4.89 |
| 4 | 220.53 | 0.01 | 504.83 | 8.38 |
| 8 | 216.77 | 0.02 | 647.48 | 13.97 |
| 16 | 225.31 | 0.02 | 733.44 | 19.44 |
| 32 | 231.74 | 0.03 | 715.57 | 33.22 |
| 64 | 234.36 | 0.03 | 674.49 | 37.62 |
| 128 | 230.38 | 0.06 | 713.39 | 55.88 |

For better comparison we provide the absolute runtimes and the absolute communication times in weak scaling experiments for all queries in Table 1.

## 7.3. Strong scaling experiments

In this section we evaluate our algorithm in strong scaling experiments, where we increase the number of nodes used to evaluate the queries on a fixed size of the database. These tests let us observe the overhead yielded by inter-node parallelization in our algorithms dependent on the number of nodes. This allows us to study the impact of using more smaller machines in comparison to using less big machines to solve fixed problems, which is a critical factor when evaluating costs and benefits of an investment in a new system.

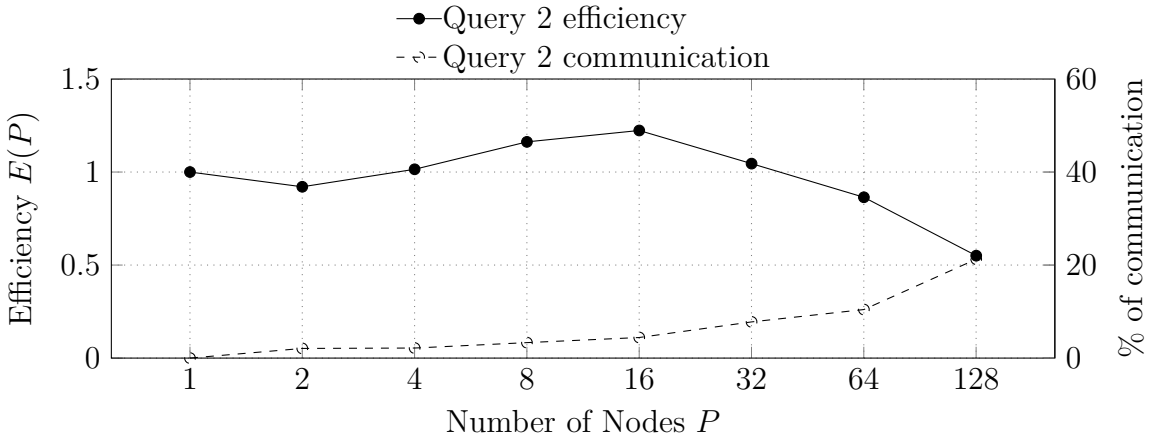


Figure 16: Query 2 strong scaling experiment. Efficiency $E(P) = T_1/(P \cdot T_P)$ and percentage of communication time on the overall runtime for different numbers of nodes $P$ and $SF = 1000$. $T_N$ is the runtime on $N$ nodes.

In Figure 16 we see that our algorithm for Query 2 keeps high degrees of efficiency when increasing the number of nodes used on the fixed-sized database. All of the local computations in the algorithm scale linearly with the size of the database which gets smaller on every node by adding more nodes. Also the amount of data to be transferred over the network gets smaller for every node as they hold less data. From 4 to 32 nodes we even see an efficiency higher than one, which might result from better cache efficiency. However, the startup overhead for communication gets higher for larger numbers of nodes while the overall runtime gets lower resulting in less efficiency for high amounts of nodes. Also the construction of the messages gets more expensive when more nodes are added to the database cluster.

While we still expect our lazy evaluation algorithm for Query 3 to scale fairly well in weak scaling experiments, we see a drastic decrease in efficiency with increasing number of nodes in Figure 17. We see that the numerous communication rounds make a larger proportion of the runtime as the number of nodes increases getting to 45% communication on the overall runtime. The reason being that the communication cost of every node is independent from the size of the input data in this algorithm, but the overall communication cost is linear in the amount of nodes used to execute the query as mentioned in Section 5.2.2. So while for other queries, the communicated volume per node decreases with increasing numbers of nodes, this is not the case for this algorithm.
In the results of our second implementation we see that, when repartitioning the tables to allow the join between *orders* and *customer* to be performed without any

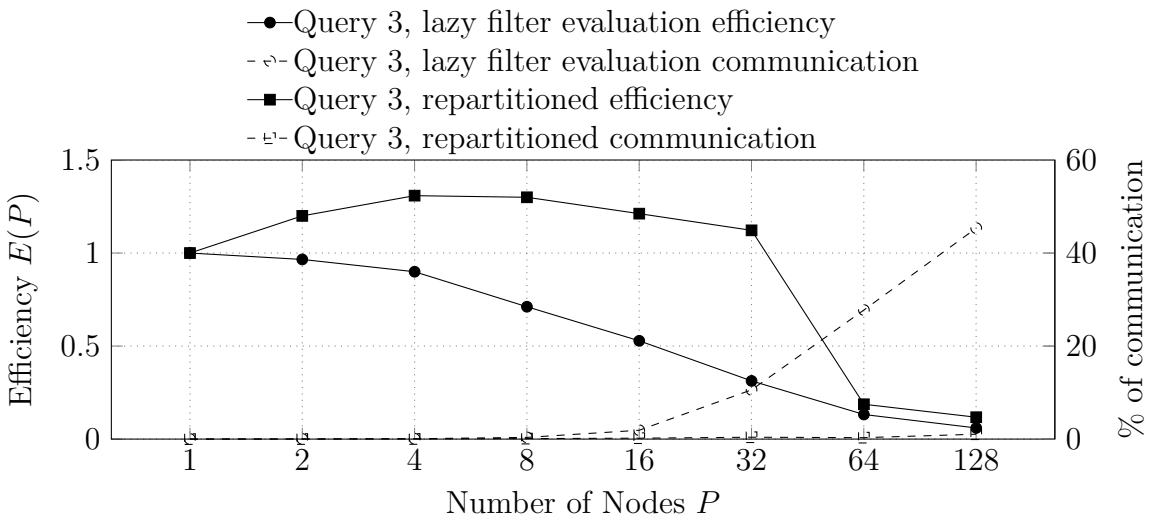


Figure 17: Query 3 strong scaling experiment. Efficiency $E(P) = T_1/(P \cdot T_P)$ and percentage of communication time on the overall runtime for different numbers of nodes $P$ and $SF = 100$. $T_N$ is the runtime on $N$ nodes.

communication, the scaling behavior is, as expected, exceptionally good and there is almost no communication taking place. The efficiency even rises above 1, which we explain by better cache efficiency. For high numbers of nodes, however, we observe a drastic decrease in efficiency. We assume this to be a result of random variations as the runtimes here are very low (see Table 2), albeit we did not find a satisfactory explanation for this behavior. Our exact implementation is just a change of a few lines of codes from Weidner [26], who also observed similar behavior.

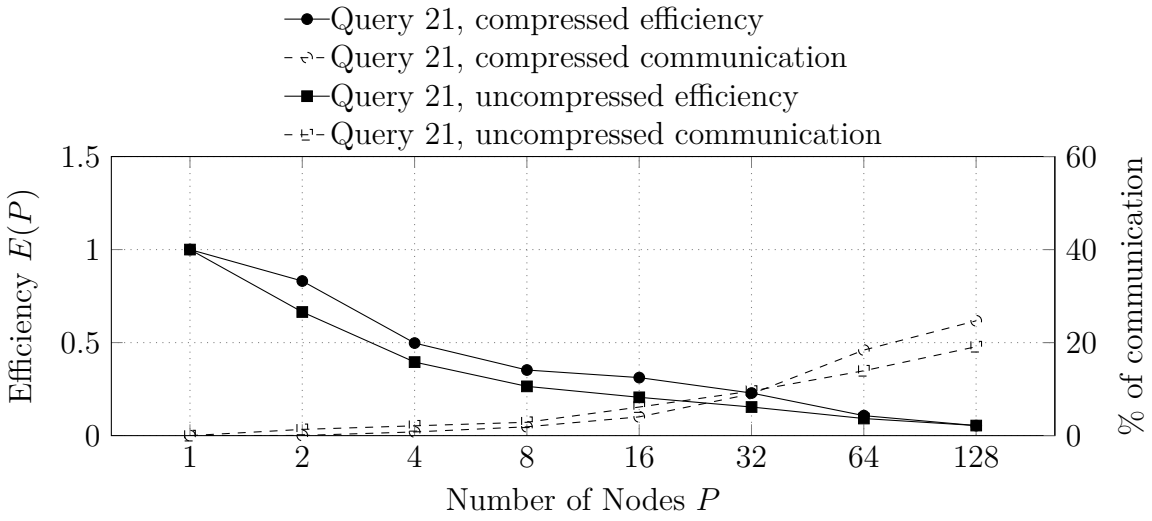


Figure 18: Query 21 strong scaling experiment. Efficiency $E(P) = T_1/(P \cdot T_P)$ and percentage of communication time on the overall runtime for different numbers of nodes $P$ and $SF = 100$. $T_N$ is the runtime on $N$ nodes.

Figure 18 shows our results for our weak scaling experiments of Query 21. We observe a decrease of efficiency with increasing number of nodes. The reason for this is again the implementation of the first step of our algorithm (see Section 7.2). As the array remains the same size for any number of nodes on a fixed sized database,

we still have to traverse over the entire array which costs a lot of time. Also, for larger numbers of nodes, communication time increases and local computation time decreases, leading to a higher relative impact of communication time on the overall runtime. We also see that for higher numbers of nodes, the uncompressed version uses less communication time than the compressed version because our compression algorithms perform better for high amounts of data which is not given when the database is split among a large number of nodes.

Table 2: Runtimes and communication times in ms for strong scaling experiments. $SF = 100$ for Query 3 and Query 21 and $SF = 1000$ for Query 2.

| Nodes | Query 2 | | Query 3 (lazy) | | Query 21 (compr.) | |
|---|---|---|---|---|---|---|
| | Overall runtime | Comm. time | Overall runtime | Comm. time | Overall runtime | Comm. time |
| 1 | 290.20 | – | 287.93 | – | 285.88 | – |
| 2 | 157.59 | 3.27 | 149.09 | 0.04 | 172.03 | 0.65 |
| 4 | 71.50 | 1.55 | 80.02 | 0.06 | 143.67 | 1.14 |
| 8 | 31.21 | 1.04 | 50.60 | 0.18 | 101.35 | 1.92 |
| 16 | 14.83 | 0.66 | 34.07 | 0.65 | 57.29 | 2.31 |
| 32 | 8.68 | 0.67 | 28.77 | 3.05 | 39.04 | 3.51 |
| 64 | 5.25 | 0.55 | 34.03 | 8.96 | 41.60 | 7.65 |
| 128 | 4.12 | 0.86 | 37.59 | 17.05 | 41.53 | 10.25 |

| Nodes | Query 3 (repart.) | | Query 21 (uncompr.) | |
|---|---|---|---|---|
| | Overall runtime | Comm. time | Overall runtime | Comm. time |
| 1 | 211.77 | – | 231.73 | – |
| 2 | 88.25 | 0.01 | 174.40 | 2.33 |
| 4 | 40.46 | 0.01 | 146.58 | 3.09 |
| 8 | 20.37 | 0.02 | 109.41 | 3.14 |
| 16 | 10.92 | 0.02 | 70.34 | 4.32 |
| 32 | 5.90 | 0.02 | 47.13 | 4.51 |
| 64 | 17.68 | 0.05 | 39.41 | 5.49 |
| 128 | 14.08 | 0.15 | 33.48 | 9.40 |

For better comparison we provide the absolute runtimes and the absolute communication times in strong scaling experiments for all queries in Table 2.

## 7.4. Comparison with other work

We compare our results to the results from Weidner [26] and to the current record holder of the TPC-H benchmark.
For the comparison to Weidner, we use scale factor 30 000 (30 TB of uncompressed data) and run our experiments on the same system.
For comparison to official TPC-H results, we chose scale factor 10 000 (10 TB of uncompressed data) where EXASolution 4.0 is the current record holder, because there are no comparable clustered results for higher scale factors that we can support. Their runtimes come from a different system than our experiments are run on, which has to be taken on account when comparing the results.
They ran the benchmark on 60 Dell Power Edge R710 with 72 GB memory each. Every node uses two hexa-core Intel Xeon X5690 QC 3.46 GHz and their linking network is the same as in our cluster.
In order to allow a best possible chance of a fair comparison we also run our experiments on 60 nodes and provide $SPEC_{intrate}$ number of the 2006 SPEC benchmark [14] like Weidner did in his work.
We provide an overview of the comparison in Table 3. Here we can see that we have better running times than EXASol by a factor of 11 to 35 and by a factor of 1.5 to 3.2 faster than Weidner, which is still a good speedup considering that those solutions were already optimized for these special queries of the TPC-H benchmark.

Table 3: Comparison to EXASolution on 60 nodes and scale factor 10 000 and Weidners results on 128 nodes and scale factor 30 000.

| | $SF = 10\,000$ | | | $SF = 30\,000$ | | |
|---|---|---|---|---|---|---|
| Query | We in [s] | EXASol | factor | We in [s] | Weidner | factor |
| 2 | 0.063 | 1.1 | 17.5 | 0.093 | – | – |
| 3 | 0.610 | 6.9 | 11.3 | 0.867 | 2.786 | 3.2 |
| 21 | 0.869 | 30.6 | 35.2 | 1.501 | 2.306 | 1.5 |
| $SPEC_{intrate}$[22] | 625 | 419 | 0.7 | 625 | 625 | 1.0 |
| Nodes | 60 | 60 | 1.0 | 128 | 128 | 1.0 |
| Total RAM | 3840GB | 4320GB | 1.1 | 8192GB | 8192GB | 1.0 |

When comparing the scaling behavior for Queries 3 and 21 with Weidners results (Figure 19), we also notice that our algorithms scale better in weak scaling experiments. While we do not achieve such low runtimes as they do for smaller numbers of nodes in weak scaling experiments, we have way better behavior with increasing numbers of nodes. This can be ascribed to the higher communication volume needed by their algorithms: In Section 5.3 we showed that our algorithm for Query 21 uses less communication volume than Weidners and the communication volume of our implementation of Query 3 is not even dependent on the database size but only on the number of nodes used because every node only needs to communicate an almost constant amount of data.
For lower numbers of nodes however, our algorithms perform worse than theirs because we decide the trade-off between local computation time and communication time in favor of communication time. As the impact of communication time is fairly

low for low numbers of nodes (or even 0 for 1 node), this does not become beneficial before a large number of nodes is used for execution. Both Queries (3 and 21) have to compute local results without the filter on the remote path in our implementation leading to more work to do. This can be seen best at the execution times for one node, where Weidner has considerable better running times than we do.

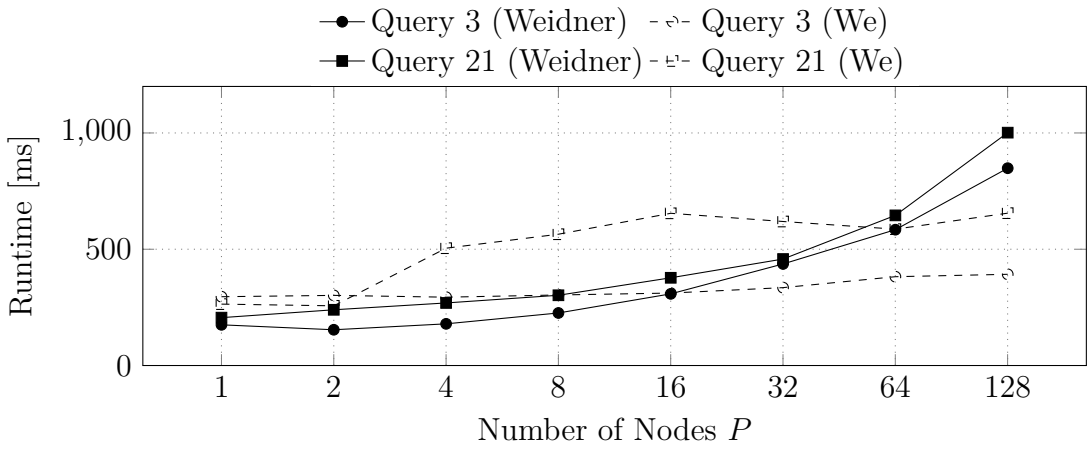


Figure 19: Comparison to Weidner [26] on weak scaling experiments ($SF = 100 \cdot P$). Implementations used for comparison: Lazy filter evaluation for Query 3 and compressed version for Query 21.

# 8. Conclusion

We showed that communication time is one of the most straitening factors of distributed query execution and presented solutions to bypass these restrictions as far as possible.

To get a deeper insight in the current state of the art, we surveyed research papers on the topics of current database technology, distributed databases and communication efficiency.

We successfully developed several algorithms to improve communication efficiency for distributed OLAP query execution and improved the techniques previous work presented. We analyzed these algorithms by using proven lower bounds on sophisticated group communication primitives and compared our algorithms to those of previous work. In order to evaluate our algorithms, we analyzed three queries of the TPC-H benchmark and applied our techniques to solve them. We then implemented all three queries in a prototype of a distributed in-memory column-stored database and ran extensive scaling tests on a large cluster with up to 128 nodes on a database with up to 30 000 GB of uncompressed data (128 000 GB for Query 2).

Our main results are techniques to find better partitionings of tables in a database and algorithms that exploit common structures of data partitionings and queries in order to reduce the total communication time used for query execution to a minimum.

We can see that our algorithms do not require much communication time when scaling the system out to support larger databases. Furthermore, for two of our three implemented queries the overall runtime did not increase too much in weak scaling experiments and at least better than previous work for all three queries. As the goal of this work was to design algorithms that allow fast execution of queries in a distributed database to support the increasing amount of data held by todays companies, we consider this a success.

However, we observe that our implementation lost efficiency in strong scaling experiments for all three queries, thus the amount of machines used for computations on smaller datasets should not be too big as most of the runtime will be yielded by parallelization overhead and communication time prevails.

In order to use our algorithms in a productive system, heuristics have to be found to determine whether our results should be applied or not. For the partitioning of the tables this can be done by hand because the tables in a database are mostly invariant and the kinds of queries that are going to be run on the tables are predictable by some degree. However, the decision whether a remote join path should be evaluated early or late via a semi-join reduction is not that easy. We provide a proven break-even point where the communicated data evens out using the selectivity of the filter conditions that can be evaluated locally. In a real-world environment this information is usually not easy to calculate. Furthermore, for the overall query runtime it is not always best to use the solution with the least communication time because the runtime is also influenced by local calculation time. It can be seen by comparison of Weidners and our results, that the local calculation time became

longer by applying the join at a later stage of the query execution. We assume this to happen in most cases, so the point at which the decision between an early and a late semi-join reduction is decided should not be the point we showed, where just communication time would be equal. If the amount for communicated data is similar for both solutions, the join should rather be solved very early in most cases. As a result of these conditions, this is not a trivial problem to solve but requires sophisticated solutions in order to boost query execution time to the best.
The algorithm that evaluates remote filters lazily can be applied without knowledge of the structure of the data. The decision whether it can be applied or not can be made entirely by knowledge of the data partitioning and should not be too much of a problem for a modern database. However, in order to minimize startup overhead for multiple communication rounds, knowledge of the selectivity of the remote filter condition can be beneficial and should be used if available.
While most current databases do not do it, it should be fairly easy to detect the possibility of saving communication volume by using information that was previously communicated like we described. As the benefits of this techniques can be quite big, as for example sending just one bit per tuple instead of a full 64 bit key, it should be used more extensively in order to boost query runtimes without changing too much of the general execution plan for a query.

Conclusive, we have developed algorithms that behave very well in realistic scaling scenarios and achieved runtimes faster than current record holders by orders of magnitudes. Further research has to be done on the techniques to find queries where our algorithms can be applied in order to integrate them into productive database systems.

# Appendix

# A. SQL queries

We provide the SQL code for the implemented queries as specified in [24].

## A.1. Query 2

Return the first 100 selected rows

```
SELECT
   s_acctbal,
   s_name,
   n_name,
   p_partkey,
   p_mfgr,
   s_address,
   s_phone,
   s_comment
FROM
   part,
   supplier,
   partsupp,
   nation,
   region
WHERE
   p_partkey = ps_partkey
   AND s_suppkey = ps_suppkey
   AND p_size = [SIZE]
   AND p_type like '%[TYPE]'
   AND s_nationkey = n_nationkey
   AND n_regionkey = r_regionkey
   AND r_name = '[REGION]'
   AND ps_supplycost = (
     SELECT
       MIN(ps_supplycost)
     FROM
       partsupp, supplier,
       nation, region
     WHERE         p_partkey = ps_partkey
       AND s_suppkey = ps_suppkey
       AND s_nationkey = n_nationkey
       AND n_regionkey = r_regionkey
       AND r_name = '[REGION]'
   )
ORDER BY
   s_acctbal desc,
   n_name,
   s_name,
   p_partkey;
```

## A.2. Query 3

Return the first 10 selected rows

```
SELECT
    l_orderkey,
    SUM(l_extendedprice*(1-l_discount)) AS revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = '[SEGMENT]'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < date '[DATE]'
    AND l_shipdate > date '[DATE]'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue desc,
    o_orderdate;
```

## A.3. Query 21

Return the first 100 selected rows

```
SELECT
   s_name,
   COUNT(*) AS numwait
FROM
   supplier,
   lineitem l1,
   orders,
   nation
WHERE
   s_suppkey = l1.l_suppkey
   AND o_orderkey = l1.l_orderkey
   AND o_orderstatus = 'F'
   AND l1.l_receiptdate > l1.l_commitdate
   AND EXISTS (
     SELECT
       *
     FROM
       lineitem l2
     WHERE
       l2.l_orderkey = l1.l_orderkey
       AND l2.l_suppkey <> l1.l_suppkey
   )
   AND NOT EXISTS (
     SELECT
       *
     FROM
       lineitem l3
     WHERE
       l3.l_orderkey = l1.l_orderkey
       AND l3.l_suppkey <> l1.l_suppkey
       AND l3.l_receiptdate > l3.l_commitdate
   )
   AND s_nationkey = n_nationkey
   AND n_name = '[NATION]'
GROUP BY
   s_name
ORDER BY
   numwait desc,
   s_name;
```

# References


[1] Bernstein, Philip A and Dah-Ming W Chiu: *Using Semi-Joins to Solve Relational Queries.* Journal of the ACM (JACM), 28(1):25–40, 1981.

[2] Bloom, Burton H: *Space/Time Trade-offs in hash Coding with Allowable Errors.* Communications of the ACM, 13(7):422–426, 1970.

[3] Bruck, Jehoshua, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal and Derrick Weathersby: *Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems.* Parallel and Distributed Systems, IEEE Transactions, 8(11):1143–1156, 1997.

[4] Chen, Ming-Syan and Philip S. Yu: *Combining Joint and Semi-Join Operations for Distributed Query Processing.* Knowledge and Data Engineering, IEEE Transactions, 5(3):534–542, 1993.

[5] Dees, Jonathan and Peter Sanders: *Efficient Many-Core Query Execution in Main Memory Column-Stores.* In *Data Engineering (ICDE), 2013 IEEE 29th International Conference*, pages 350–361. IEEE, 2013.

[6] DeWitt, David and Jim Gray: *Parallel Database Systems: The Future of High Performance Database Systems.* Communications of the ACM, 35(6):85–98, 1992.

[7] EXASOL: *EXASolution / Built for speed, without limits.* http://www.exasol.com/en/products/exasolution/. [Online; accessed 11-September-2014].

[8] Färber, Franz, Norman May, Wolfgang Lehner, Philipp Grosse, Ingo Müller, Hannes Rauhe and Jonathan Dees: *The SAP HANA Database–An Architecture Overview.* IEEE Data Eng. Bull., 35(1):28–33, 2012.

[9] Fraigniaud, Pierre and Emmanuel Lazard: *Methods and Problems of Communication in Usual Networks.* Discrete Applied Mathematics, 53(1–3):79 – 133, 1994.

[10] Garcia-Molina, Hector and Kenneth Salem: *Main Memory Database Systems: An Overview.* Knowledge and Data Engineering, IEEE Transactions, 4(6):509–516, 1992.

[11] Golomb, Solomon W.: *Run-Length Encodings.* In *IEEE Transactions on Information Therory*, pages 399–401. IEEE, 1966.

[12] Gonzalez, Teofilo and Sartaj Sahni: *Open Shop Scheduling to Minimize Finish Time.* Journal of the ACM (JACM), 23(4):665–679, 1976.

[13] Hailey, Kyle: *SQL joins visualized in a surprising way.* http://www.oraclerealworld.com/sql-joins-visualized-in-a-surprising-way/, August 2013. [Online; accessed 8-September-2014].

[14] Henning, John L: *SPEC CPU2006 Benchmark Descriptions.* ACM SIGARCH Computer Architecture News, 34(4):1–17, 2006.

[15] Kemper, Alfons and Thomas Neumann: *HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots.* In *Data Engineering (ICDE), 2011 IEEE 27th International Conference*, pages 195–206. IEEE, 2011.

[16] Koutris, Paraschos: *Bloom Filters in Distributed Query Execution*, 2011.

[17] Plattner, Hasso: *A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database*. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2. ACM, 2009.

[18] Putze, Felix, Peter Sanders and Johannes Singler: *Cache-, Hash- and Space-Efficient Bloom Filters*. In *Experimental Algorithms*, pages 108–121. Springer, 2007.

[19] Rödiger, Wolf, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, Thomas Neumann and CA San Mateo: *Locality-Sensitive Operators for Parallel Main-Memory Database Clusters*. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference*. IEEE, 2014.

[20] Sanders, Peter, Sebastian Schlag and Ingo Müller: *Communication Efficient Algorithms for Fundamental Big Data Problems*. In *Big Data, 2013 IEEE International Conference*, pages 15–23. IEEE, 2013.

[21] Sanders, Peter and Jesper Larsson Träff: *The Hierarchical Factor Algorithm for All-to-All Communication*. In *Euro-Par 2002 Parallel Processing*, pages 799–803. Springer, 2002.

[22] SPEC: *All SPEC CPU2006 Results Published by SPEC*, 2013. [Online; Accessed on May 17, 2013].

[23] Stonebraker, Mike, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil et al.: *C-Store: A Column-oriented DBMS*. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[24] Transaction Processing Performance Council (TPC): *TPC BENCHMARK™ H, Standard Specification*, 2012.

[25] University, Michael Stonebraker and Michael Stonebraker: *The Case for Shared Nothing*. Database Engineering, 9:4–9, 1986.

[26] Weidner, Martin: *Execution of OLAP Queries in Distributed Main Memory Databases Based on TPC-H*. Master's thesis, Karlsruhe Institute of Technology, 2013.