Code generation for parallel plan execution in a big data database architecture

Stephan Pfistner

June 3, 2013

Abstract

Large scale data analysis has become one of the key problems of modern world technological systems. Diverse systems exist to overcome this problem, but still suffer systemic flaws.

This paper documents work on code generation for a elastic and robust database management system called XDB which tries to tackle these problems from a non-typical direction. The main focus of lies on generation of plans for highly parallel and fast data processing while being robust in execution. Therefore the structure of database management systems is examined with regards to plan generation and parallelization, required parts of an implementation for the XDB system are conceived and implementation is documented.

Contents

List of Figures

List of Listings

1 Motivation

1.1 Problem statement

The publication of Google's ground breaking MapReduce paper revolutionized large scale data processing [5]. Its open-source implementation Hadoop has since become the de facto standard for analysing Big Data [13]. Despite its advantages like scalability to thousand of nodes and robustness of the execution model, Hadoop is inherently inefficient[1].

In contrast to other work, which focuses on providing solutions to the shortcomings of Hadoop, the elastic, robust and flexible database management system XDB tries to solve this by combining the major benefits of traditional relational databases and Hadoop. As XDB is a middleware for existing databases like MySQL or PostgreSQL, it benefits from the maturity and efficiency in query processing of the underlying database systems while exploiting the scalability and robustness of concurrent execution on multiple nodes required for big data analytics.

All but its data persistency features had to be developed from ground up with attention to its characteristics like distributed data storage and execution and flexible programming model [3]. Part of such this system is code generation to map functional frontend query input into individual jobs for data execution while being able to match the robustness needed in an highly distributed environment. This includes parallization to handle the distributiveness of the data sources and provide speed-up where possible to be competitive to other systems like Hadoop.

Thus, Code generation in XDB requires non-standard but efficient approaches for code generation and attention to the specific needs of a distributed database middleware which will be examined in this paper.

[1]See section 2.2 for explanation on this statement.

1.2 Goals

Goal of this paper is to provide an overview of the work done during the last view months on code generation and parallelization in the distributed database system XDB. Insight into specific techniques will be given and the resulting implementation will be presented and discussed.

2 Background

To understand the decisions made while developing code generation and parallelization in XDB, knowledge about functioning of other well-known database systems is required.

The following subsections will give an overview of standard techniques used for code generation and parallelization in relational database management systems and Hadoop. MapReduce and its implementation Hadoop are explained. Finally the framework of XDB will be explained in detail and code generation and parallelization will be located in this system.

2.1 Relational database management systems (RDBMS)

Relational database management systems make use of the relation database model, presented in 1970 by Edgar Codd, to manage data. They are easy to understand and to use and thus became the predominant choice for electronically storing structured data.

Data is managed in relations (tables) which contain data sets. Each tuple (row) of a table is a data record with the same kind of attributes (columns of the table). The relational schema defines the number and types of attributes of the table.

Queries against the relational database are expressed in a query language based on a relational algebra. The most commonly used one is the Structured Query Language (SQL). The original relational algebra by Codd defined eight relational operators. [11]

These are:

Union - Combines tuples of two relations and removes duplicates from the result.

Intersect - Produces the set of tuples two relations have in common.

Difference - Produces the set of tuples two relations do *not* have in common.

Cartesian product - Results in a set of tuples which includes every tuple of a relation combined with every tuple of another relation.

Selection - Retrieves all tuples from a relation which meet a specific criteria.

Projection - Extracts only specific attributes from every tuple in a relation.

Join - Two relations are connected by a set of common attributes. Each match from a relation to another is added to the result relation.

Relational division - Partitions a relation (divisor) using the tuples of another relation (dividend) and is effectively the opposite of the cartesian product.

Relational databases use many subsystems to handle user queries. Most commonly the work flow can be described briefly as follows: Using a query compiler the user query is parsed into a query plan which is then optimized. The plan expresses a sequence of actions the RDBMS will perform to answer the query. It is then passed to the execution engine which requests small pieces of data, typically tuples of a relation, from a ressource manager that knows about the data files holding the relations. It uses indices to find elements more quickly. Then there is a buffer manager capable of supplying the data from secondary storage.[2][3]

Because of the sequential processing of queries, possibilities to make use of massive parallel systems for relational database systems are limited:

[2]In this context *storage* normally means hard disk.

[3]For a full overview of subsystems used to handle queries, see [11, pp. 10]

Multiple queries can be trivially run in parallel, if they are independent.[4] Limiting resource in such cases are disk I/O and processing power. [11, pp. 403]

Missing processing power can be addressed by using better algorithms if available, or scale up by upgrading the database system's hardware. Disk I/O on the other hand can be speed up by using multiple disks (split or mirror data), but this holds only true for throughput and not for response time.[5] In relational database systems indices can be used to lower response time and increase throughput as they enable the system to address data more efficiently and reduce the amount of data needed to be transferred from secondary storage.

All mentioned methods are used to speed up the serial processing of queries. To effectively use multi- and many-core systems relational databases can make use of *horizontal partitioning*. This enables the database system to even process single queries in parallel.

Horizontal partitioning involves spreading the table's rows across several logical tables by applying a certain partition criteria. The union of all partitions provides a complete view of all rows. Commonly used partition criteria are:

Range partitioning - Select the target partition for a particular row based on the partitioning key being inside certain value ranges.

List partitioning - Partitions are assigned lists of fitting values. The partition is chosen by matching the partitioning key with a partition's list.

Hash partitioning - A hash functions determines the partition.

As each partition is a logical independent table, queries on which are limited on particular partitions are also independent. Thus, the initial statement can be applies: Independent queries can be executed in parallel.

[4]Problems of transactional processing and query interaction will not be considered in this paper.

[5]Both throughput and response time are important variables for examining the performance of a system.

This can even be used by the database system when processing queries on the whole dataset: Queries with operations on partitioned relations can be split up by specific rules and run in parallel on each partition. The result of the whole operation is the union of all child operations on the partitions.

Despite such improvements, limitations of the database system as a whole still apply. Because up-scaling of a single system is limited, other mechanisms for scaling out have to be used:

- Scaling out via *sharding* involves partitioning databases into parts ("shards") and locating them on different database systems. Each system is then responsible for managing access to shards located in it. This improves performance as each database system can be placed on different hardware, and thus the database can benefit of the accumulated performance (read and write) of all machines. Furthermore it allows further dynamic scale-up by creating or dismissing shards if needed, however cost-intensive repartitioning is required. Its downsides are its inflexibility if no automatic system for distribution is used and its difficulty to maintain integrity. Also, sharded databases are often not fault-tolerant to loss/corruption of single shards.

- Another common method for scaling out is to replicate the data across multiple servers. Each system contains the same data and queries are split out across all systems. This can vastly improve read performance, as the load is distributed but does not improve write access: To maintain integrity data is replicated from one master node to multiple slaves, which means there is still only one node for managing write access. This method scales well for environments with a high number of fast reads and low number of writes/updates, like web sites [9]. It is fault tolerant, maintains integrity and allows adding more slave nodes as needed, which can increase the load on the master node during replication. This method increases the throughput by raising the number of active queries at a moment, but does not affect the response time of each query as the procedure to answer the query is still the same as on a single system database management system.

There are proprietary solutions to provide scaling out in a relational database system, like Microsoft's SQL improvement SCOPE. It abstracts the query from the technical difficulties present to make use of parallelization [4] [14] [15].

2.2 Hadoop and Hive

The performance gains achieved by scaling relational database systems are sometime not enough for data intensive and/or distributed applications, thus special database systems have been developed to handle requirements.

In 2004, Google published a paper describing a novel parallel programming model called MapReduce, specially suited for massively parallel processing huge amounts of data. The MapReduce work flow splits data processing into two main steps: *map* and *reduce*. The map procedure performs filtering and sorting of input data whereas the reduce procedure performs a summary operation on map's results. The MapReduce framework orchestrates this process by splitting input into sub-problems, distributing the tasks across servers and managing communications and data transfers between them. It also provides redundancy and thus fault tolerance of the whole process by being able to restart certain steps of the operation if workers fail.

Map operations can be distributed for processing and each processed in parallel. In practice this is limited by the rate of which data sources can be divided into smaller sub-problems. Reduce operations can be run in parallel if all map results with the same key are presented to a single reducer at the same time, or the reduce function is associative and thus can be applied incremental. Because of the scalability of map and reduce, the whole MapReduce system is able to massively scale out and provide huge data throughput. In fact its ineffective for small data sets, but can be used to process significantly larger datasets than common relational database systems could. Many common real world operations can be expressed using this model. [5]

Soon after publication of Google's paper developers wanted to use the novel programming model in their own systems, thus a open-source Java-based implementation named Apache Hadoop was born at Yahoo to power their search. It is not only based on MapReduce but also on Google's paper about its scalable distributed file system called Google File System. Hadoop's correspondent implementation was named Hadoop Distributed File System (HDFS).

Hadoop is a software framework to support data-intensive distributed applications. It implements the MapReduce paradigm by splitting workload into small fragments which can be executed and re-executed on any node in the cluster. Hadoop can make use of different file systems, but they should be location aware: Hadoop uses information about the physical location (connection to network switches) of nodes to store data on or near computing nodes to provide very high aggregate bandwidth across the cluster. HDFS uses this information in addition to keep copies of data on different racks to reduce impact of local hardware failures and provide fault-tolerance on file system level.

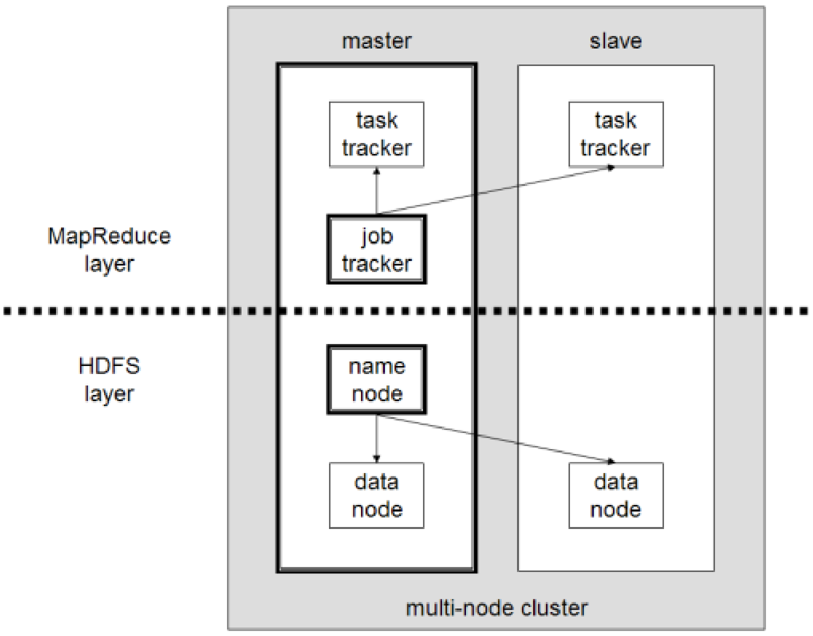


Figure 1: Hadoop architecture [8]

Figure 1 shows the architecture of a multi-node Hadoop cluster. A small cluster will include a single master and multiple worker nodes. The master consists of a JobTracker, TaskTracker, NameNode and a DataNode. A worker node acts as both a DataNode and TaskTracker, thought it is possible to have data-only worker nodes and compute-only worker nodes. Hadoop requires the Java Runtime Environment and Secure Shell to be set up. Larger clusters use a dedicated machine as NameNode to manage the HDFS file system index. Technically, a complete Hadoop cluster can be run a single machine, but that would render the fault-tolerant features useless.

Because of the advantages over traditional parallel database systems like its ability to scale to thousands of nodes, its robust execution model and the support of the MapReduce paradigm Hadoop became the de facto standard for analysing big data. Many high-level optimizable programming languages supported MapReduce and could be used to compose MapReduce programs. To simplify the switch of paradigms from relational database models to the schema-less parallel MapReduce model extensions features extensions like Apache Hive were developed.

Apache Hive is a data warehouse infrastructure built on to of Hadoop to provide data summarization, query and analysis [12]. It supports large datasets stored in Hadoop-compatible file systems and provides a SQL-like language (HiveQL) while supporting MapReduce. To reduce the response time of queries it provides indices.

While highly scalable, Hadoop is inherently inefficient [3]:

- First, Hadoop materializes each intermediate result of a map or a reduce function either locally or into its distributed file system (HDFS). This execution model results in a poor single query performance with high resource consumption.

- Second, data locality is a huge problem since the storage layer is separated from the computation layer: When data is loaded in HDFS it is randomly partitioned often resulting in expensive operations, which copy data over the network.

- Third, efficient operations as well as index structures to read data selectively from disk are missing. Hadoop only provides unary operators and needs to read all data from disk. Thus, the first map function typically selects the relevant data by scanning all input data. Moreover, join operations are unnatural in Hadoop since binary operators are not supported, leading to inefficient workarounds.
- Finally, Hadoop only allows a strict sequence of map and reduce operations while intermediate results are re-partitioned after each map-operation (even if it is not needed).

However, software to provide solutions to the shortcomings of Hadoop has been developed (e.g. Hadoop++, HAIL, HadoopDB).

2.3 The XDB approach

The XDB database management system tries to combine the benefits of relational databases and Hadoop. Its main goals are to provide elastic optimization and execution, robust execution-plans and a flexible programming model. While databases provide efficient techniques for query execution that have matured over many decades, they lack scalability and robustness properties that are provided by Hadoop for big data analytics. SQL makes it difficult to express advanced analytical tasks or to use unstructured data, whereas the flexibility of the MapReduce programming model is one of Hadoop's strengths.

XDB's approach to achieve its goals tries not to fix the problems of Hadoop: Its a middleware for existing relation database systems like MySQL or PostgreSQL in order to benefit from the maturity of the underlying database and its efficient query processing techniques such as efficient relational operators and different types of indices. The databases remain unchanged, XDB only works on top of them: XDB uses multiple instances of the database systems to execute its queries.

To provide elastic optimization and execution XDB incrementally optimizes and executes its execution-plans. As small errors in cardinality estimations during optimization can have high impact on execution and the available resources for a job are dependent on other users workloads, incremental execution and re-optimization tries to adopt the plans for fast execution during runtime.

Robust execution of queries is achieved by splitting the execution-plan into independent sub-plans which materialize their result for recovery after execution. In case of node failures sub-plan execution can easily be reassigned to other nodes. A cost-based optimizer decides about when to trade execution time for robustness and vice versa. This allows a fine-grained and fault-tolerant execution in opposite of the all-or-nothing query principle in relational database.

A flexible programming model is provided by extending SQL's "one-query one-result" principle with purely declarative functions with multiple in- and outputs.[6] Furthermore user defined functions with optimization rules are introduced.

Figure 2 show the XDB architecture. As XDB is based on existing database systems the lowest layer is the database layer. The layers above form the middleware layer, which extends the mere databases by the mentioned properties: elasticity, robustness and flexibility.

The workflow of processing a query is as follows: The compile server accepts a program written using the FunSQL programming model. It then translates the program into a compile-plan, which is basically a data-flow graph consisting of relational operators (resulting from declarative functions). The compile-plan is then annotated with catalog metadata. Before transferring the plan to the master tracker the plan is optimized by applying a set of rule-base optimizations. The master tracker is responsible to manage and monitor cluster resources and assign compile-plans to plan tracker. The plan tracker is then responsible for executing and monitoring a compile-plan. Therefore it divides the compile-plan into partial sub-plans. Each of them is parallelized and executed individually. By materializing their results robustness and adaptive parallelization are achieved: The tracker decides about the degree of parallization once all inputs of a sub-

[6]It is a standalone functional language called *FunSQL*, see [2].

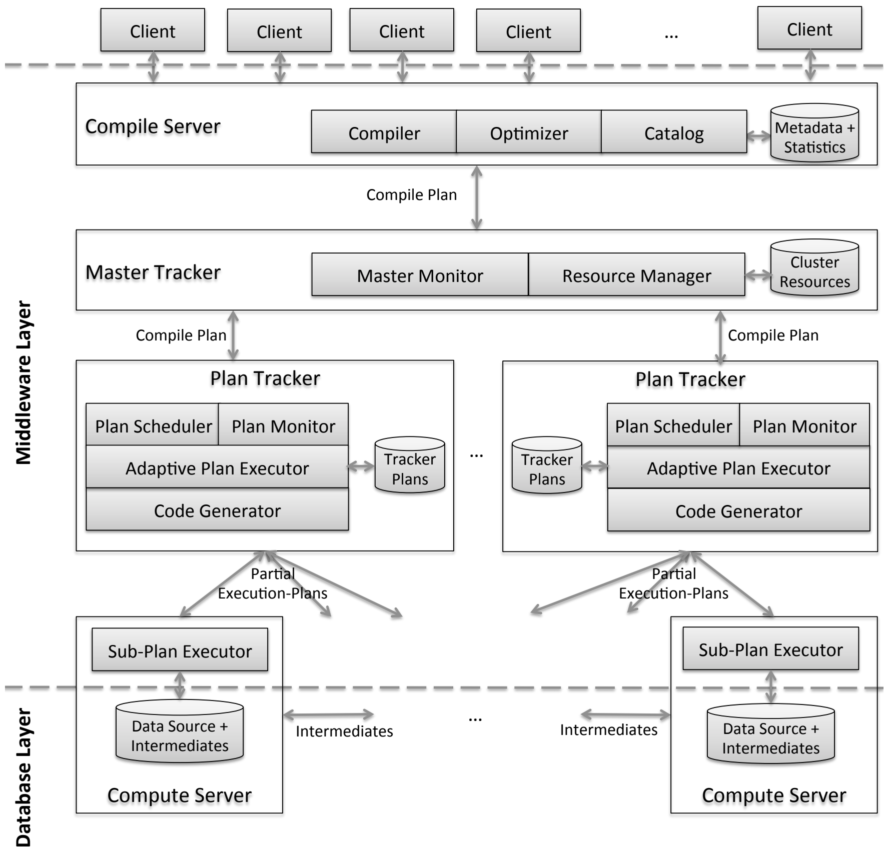


Figure 2: XDB architecture [3]

plan are available. To execute a sub-plan it uses the code generator to create execute plans with code suitable for the underlying database system and deploys that to a compute server. Each compute server host an instance of a underlying database on which they execute assigned plans and a sub-plan tracker which tracks all plans executing on that node. The scheduler tries to assign plans location aware, meaning it tries to execute plans on compile server containing the data sources needed in the plan. Results are then materialized to the underlying database (e.g., memory tables in MySQL). This has the advantage that indexes and partition functions can be leveraged for the intermediate results. To access data stored on other nodes XDB uses the remote tables concept present in most relational database systems, e.g. the federated engine in MySQL and dblink in PostgreSQL.

This approach can be seen as a combination of concepts used in traditional relational database systems and Apache Hadoop: XDB exploits the effectiveness of relational database systems in query processing by using them as a solid basis for execution. It then tries to achieve the robustness of Hadoop by using the "distributed data source and computation" and the "store interim result" approaches. As queries are possibly executed on multiple systems scalability is should also be provided.

Code generation is a key point to achieve this goals and keep up a decent speed by providing parallelization.

3 Analysis and Concepts

Code generation works as a point of connection between the abstracted XDB compile plans and the underlying database system. It's not only responsible for transforming plans between them, but to include parallelization and provide information for later elastic adoption during execution. This information can be though of as a kind of "predetermined breaking point" for splitting the plans when necessary.

To enable the system to execute plans (serial and parallel) and adopt dynamically during execution the code generation layer has to execute multiple steps: First, the compile plan created by the compiler has to be annotated to allow the following tasks to use operator-specific information. This includes decisions about splitting the plan into sub-plans. The next step is to parallelize the compile-plan if needed. Next, the plan should be optimized by the code generation system to allow faster execution on the underlying database system.[7] Then the actual code generation happens: According to the annotated sub-plan information a query tracker plan with tracker operators for each sub-plan is generated. The tracker operators contain code suitable for the underlying database system (e.g. SQL for MySQL databases). This includes not only the code used for execution but in- and output DML statements and further data taken from the corresponding

[7]This is then the overall second optimization, targeting the execution speed of the later generated code.

compile plan operators. Using the scheduler the completed query tracker plan is then executed and thus the operators are transferred to sub.plan executors which will directly use the generated database code.

The subsequent subsections will describe some these code generations aspects in detail and contain reasoning about decisions made while developing them.

3.1 Annotation

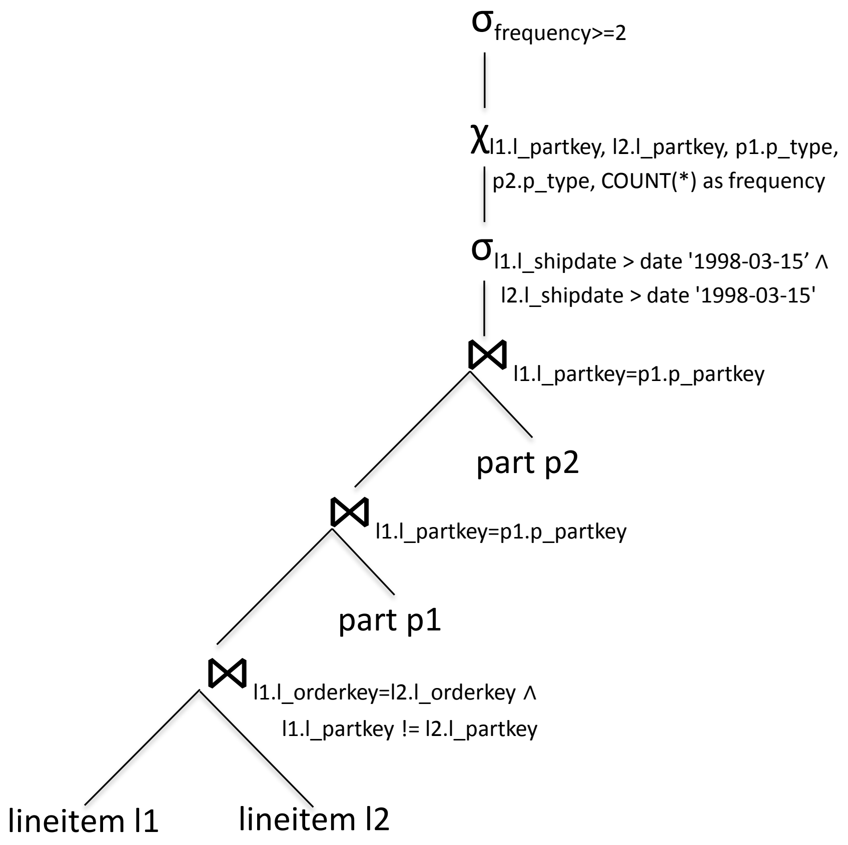

σ frequency>=2
χ l1.l_partkey, l2.l_partkey, p1.p_type, p2.p_type, COUNT(*) as frequency
σ l1.l_shipdate > date '1998-03-15' ∧ l2.l_shipdate > date '1998-03-15'
⋈ l1.l_partkey=p1.p_partkey
part p2
⋈ l1.l_partkey=p1.p_partkey
part p1
⋈ l1.l_orderkey=l2.l_orderkey ∧ l1.l_partkey != l2.l_partkey
lineitem l1
lineitem l2

Figure 3: Compile plan

At the beginning the code generator receives a compile plan like the one in figure 3. It contains information about parent/child-relations between the operators and some basic connection information for table operators (that initial read data from the database system on execution). To further process the plan additional information for each operator is needed, thus annotation is the first step in code generating.

To allow plan translation and elastic adoption the mentioned predetermined breaking points have to be set. They specify if the interim result of a operator should be persisted (“materialized”). A rule set can be created to determine the state of this flag for each operator:

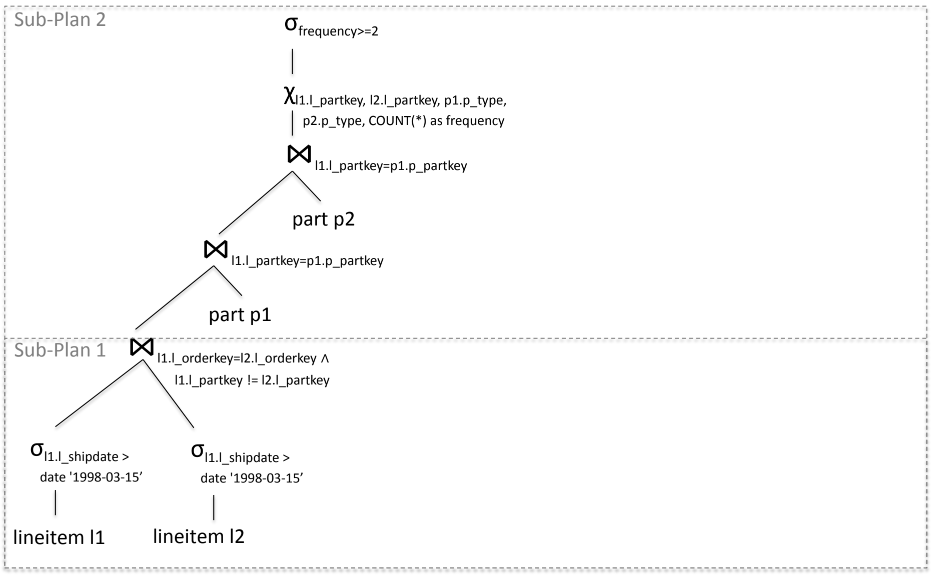
- The first obvious rule is to split on a absolute requirement to make further code generation work. This is case for the last operator of a plan. Its result apparently has to be materialized as it should be readable by the client as a result of the whole query, even after completion of this operator.

- Furthermore a split is required for operators which have multiple “parents”, meaning multiple operators whom input depends on this operator. Such constructs are just not mapable on relational databases with a single query. This operator else had to be run multiple times, one time for each parent, which would result in a performance killer. To prevent the results of such operators are materialized and each parents reads its input from this materialization. This also supports scalability as the parents even could be run on different nodes.

To allow the following systems to determine the result of this operation the state of the materialize flag is annotated on each operator, specifying where a plan can or has to be split into sub-plans.

The materialize flags effectively determines the borders of sub-plans: A sub-plan of a compile plan consists of all operators below a operator with the materialized flag set. The materialize-annotated version of the example from above can be seen in figure 4.

The annotation step also persists information about partitioning and connection association for operator deployment on execution. To keep data traffic between XDB nodes low and speed high, the XDB scheduler tries to execute queries near their data sources. To achieve this, it needs information about where the data on which a operator depends is located. Partitioning information is needed for parallelization.



Sub-Plan 2
σ frequency>=2
χ l1.l_partkey, l2.l_partkey, p1.p_type, p2.p_type, COUNT(*) as frequency
⋈ l1.l_partkey=p1.p_partkey
part p2
⋈ l1.l_partkey=p1.p_partkey
part p1
Sub-Plan 1
⋈ l1.l_orderkey=l2.l_orderkey ∧ l1.l_partkey != l2.l_partkey
σ l1.l_shipdate > date '1998-03-15'
σ l1.l_shipdate > date '1998-03-15'
lineitem l1
lineitem l2

Figure 4: Compile plan with sub-plans

At the beginning of code generation this information is only available for table operators, as their data location is the location of the tables they are referring to. The operators using them as data sources obviously inherit their data location. To avoid expensive repartitioning, partition information are also inherited. Edge cases are operators depending on multiple other operators with different data locations or partitions: A decision about the inheritance of has to be made. A simple move is to statically inherit one of the childs, a better approach seems to be a cost-base decision (e.g., inherit the child with the *most* tuples) to keep computational cost and traffic between nodes as low as possible.

The data source annotation is not set statically, but just a starting point for the executor. Depending on the final deployment of operators on nodes during runtime the decision about the best fitting node can and will change if needed.

This annotations are optimized for a maximum of performance during execution, but that's is not always desired. Thus, the annotation step includes the possibility to switch to a more robustness oriented annotation model. It may materialize more often to allow restarts of operators with less overhead.

3.2 Parallelization

After annotation the compile plan contains all information needed to be parallelized. This is a pure optional, but advised step. The idea behind parallelization of compile plans lies in acceleration of execution.

Parallelization splits plans at specific points and copies the newly created sub-plans (all operators following downwards). Without further adjustment this has no benefit, it only makes sense if parallelization of computations are possible at all. This is the case if using multiple partitions on the data sources, which is a similar approach as used in horizontal partitions by relational database systems presented in section 2.1: The parallelized sub-plan can be run on each partition independently and thus in parallel. The only requirement for parallelizing to work is that all data sources have to have a common partition scheme[8] applied to the same partition key. If not the result is indefinite and likely wrong as operations on partition key are likely not to match as expected. The results of the last operators of all parallelized sub-plans have to be aggregated at last to create the full result.

The sub-plan is copied n times for n partitions present in the data sources. Each parallelized sub-plan then executed only on one partition of the data sources. This results in a substantial speed benefit during execution as the execution time of operations can be split by nearly n for an optimal case.[9]

To parallelize several steps have to be taken:

1. Partition candidates have to be found. Depending on the information from annotator's edge case decisions multiple possible partition schemes for each operator could be realised: The annotator decided

[8]A common partition scheme means the same partition criteria with the same number of partitions.

[9]Static overhead for execution additional sub-plans and aggregation of interim results apply and thus lower the theoretical benefit of n.

locally by examining only the operator and its children. The parallelizer now appends *all* possible partition schemes to the parent operator to later decide globally on what partition schemes should be used.

2. Generate possible plans. Partition candidates of the plan have to be combined to form complete plans with less as possible partition conversions needed between operators. This generates several possible plans.

3. Chose plan. From all the possible plans generated one has to be chosen as final. This decision is made by examine each plans cost based on estimations about number of tuples and selectivity of operators.

4. Generate parallel plan. If found sub-plans below oprator with multiple partitions are copied. Each sub-plan then execute it's operations only on one partition. At last a method have to be found to recombine the results of the sub-plans to extract the result.

The result of this process is a parallelized plan, possible with sub-plans, as seen in figure 5. Under the precondition that the tables *l1*, *l2*, *p1* and *p2* all have two partitions and are partitioned by the same criteria on the same key two sub-plans with two executions each can be found. Both sub-plans identified during the annotation step resulted in two parallel sub-plans each.

3.3 Optimization

The first optimization run done by the compile server focusses solely on an efficient compile plan including push down of operators. After parallelization a very fast compile plan should have been created, that however is very fragmented. This has a negative influence on the execution speed because of a high static/dynamic cost ratio during execution: For every small operator a new execution slot has to be allocated, the operator has to

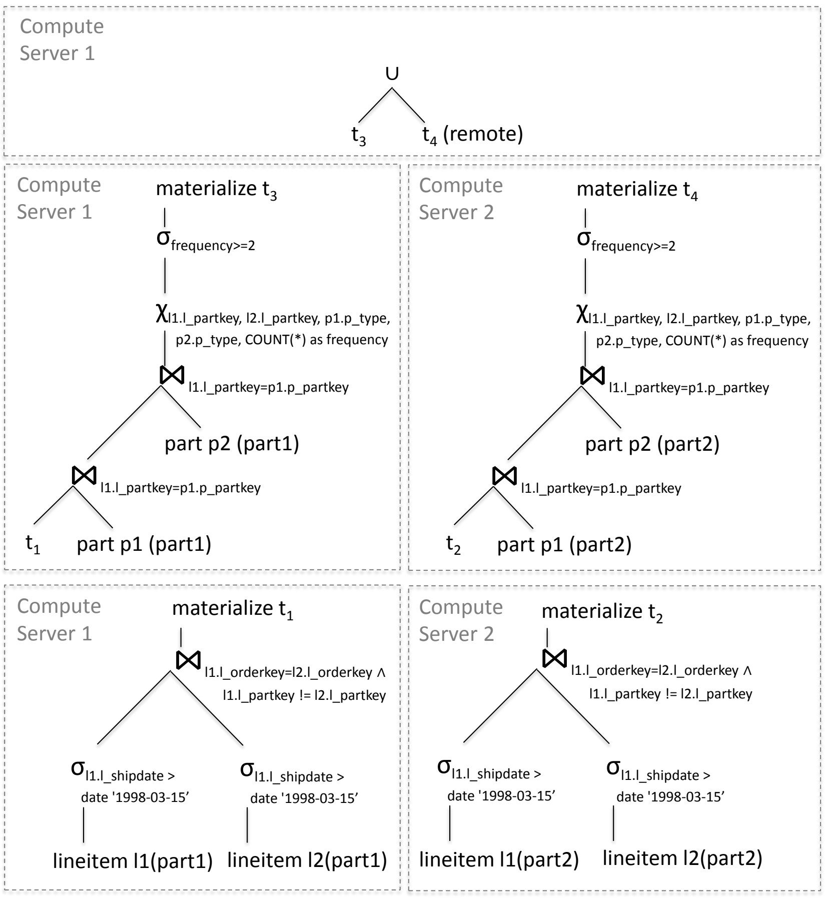

Compute Server 1
∪
t_3
t_4 (remote)
Compute Server 1
materialize t_3
$\sigma_{frequency>=2}$
$\chi_{l1.l\_partkey,\ l2.l\_partkey,\ p1.p\_type,\ p2.p\_type,\ COUNT(*)\ as\ frequency}$
$\bowtie_{l1.l\_partkey=p1.p\_partkey}$
part p2 (part1)
$\bowtie_{l1.l\_partkey=p1.p\_partkey}$
t_1
part p1 (part1)
Compute Server 2
materialize t_4
$\sigma_{frequency>=2}$
$\chi_{l1.l\_partkey,\ l2.l\_partkey,\ p1.p\_type,\ p2.p\_type,\ COUNT(*)\ as\ frequency}$
$\bowtie_{l1.l\_partkey=p1.p\_partkey}$
part p2 (part2)
$\bowtie_{l1.l\_partkey=p1.p\_partkey}$
t_2
part p1 (part2)
Compute Server 1
materialize t_1
$\bowtie_{l1.l\_orderkey=l2.l\_orderkey\ \wedge\ l1.l\_partkey\ !=\ l2.l\_partkey}$
$\sigma_{l1.l\_shipdate\ >\ date\ '1998-03-15'}$
$\sigma_{l1.l\_shipdate\ >\ date\ '1998-03-15'}$
lineitem l1(part1)
lineitem l2(part1)
Compute Server 2
materialize t_2
$\bowtie_{l1.l\_orderkey=l2.l\_orderkey\ \wedge\ l1.l\_partkey\ !=\ l2.l\_partkey}$
$\sigma_{l1.l\_shipdate\ >\ date\ '1998-03-15'}$
$\sigma_{l1.l\_shipdate\ >\ date\ '1998-03-15'}$
lineitem l1(part2)
lineitem l2(part2)

Figure 5: Parallelized compile plan [3]

be transferred, output tables (even if they mostly in memory) have to be allocated and finally the query has to be executed.

This optimization run tries to lower this ratio by reducing the number of operators needed to be executed based on combination of equal operators by specific rules:

Multiple joins Join operators with direct parent-child relationship can be easily combined by adding the join tokens and conditions to the parent, redirect all children s of the child to the parent and delete the child-join.

Join after unary Unary operators and child joins can be combined by replacing the parent unary operator with the child join and adopt all operations from the unary operator.

Multiple Unary Operators Multiple unary operators in a direct row of control flow can be combined by accumulating all their operations into one operator.

After applying this optimizations the processing of the compile plan is completed.

3.4 Plan translation

The final step in code generation is to translate the compile into operations suitable for the underlying database system. Based on the work of the previous steps this is rather uncomplicated: For each extracted sub-plan a corresponding operator is created, accumulating all its functionality. For SQL based databases like MySQL and PostgreSQL this is the mere task to stack the SQL representations of each compile plan operators into another, leaving wildcards for variables like source and destination tables.

All the this generators are packed into a query tracker plan using the same control flow as the sub-plans of the compile plan used. Now, the plan is ready for deployment and execution.

4 Implementation

This section described the implementation of the steps described in section 3.

4.1 Basic concepts

A basic concept implemented numerous times in XDB is the visitor pattern. It separates the algorithm from the object structure on which it operates and allows to add new operations without modifying the existing. A visitor implements a algorithm using different functions, one for each object is should be able to operate on. The processing is kicked off by calling a object's accept function with the desired visitors which then calls the visitors visit function with itself as a reference. [7, pp. 525]

XDB uses the visitor pattern especially to walk through compile/query tracker plans and operate on each instructions. There are implementations available for each bottom up and top down visitors.

4.2 Annotation

The annotation step is completely handled by visitors: Each annotation model (e.g. performance or robustness) has its own implementation. Decisions about materializations are implemented as a simple boolean flag to represent the current state. The presented rule set of deciding about it can be condensed into three simple lines of code as seen in listing 1.

List of Listings 1: Materialize flag annotation

```
if (op.getParents().size() != 1) {
  op.getResult().setMaterialized(true);
}
```

Partitioning and connection associations are just copied from the children.

4.3 Parallelization

The implementation of the parallelization step differs from the described theory as the last step comes first: To combine the results of the parallel executions a new operator is introduced named data exchange. The data exchange operator is responsible to combine different results into one result set and adjust partitioning as needed. Its representation on execution corresponds to the SQL *UNION ALL* operator. Data exchanges are inserted above every join and aggregation operator as they often a parents of a sub-plan and contain partitioned data sources.

The next step is parallelize the plan as depicted in section 3.2:

1. To determine partition candidates each operator is examined using a visitor. Possible partition candidates are no partition and any of the partitioning of the children of this operator. They are added to a list on the operator. As only operators directly reading from table have partitioning information annotated at the beginning its partitions transmigrate through the plan.

2. Possible plans are generated by aggregating a superset of all partition candidates of all operators of the plan. For each of them a new copy of the current plan is established and annotated with the partition information of that candidate. This is done by visiting the plan bottom up and inheriting the partition information of the child. Unneeded data exchanges are removed and remaining are set to correctly combine the partitions of the children.

3. Choosing the best plan is currently done by a simple heuristic based on estimated costs for each plan: For each removed data exchange during the plan generation phase the cost c of the plan is increased by the following formula: $1/(10 * \text{number operators})$. This calculation is done for every plan and the cheapest is chosen at the end.

4. To generate the parallel plan, the parallel sub-plans are deep-copied and added to the plan. That means, that all object and object references are duplicated.

Apart from the planned parallelization steps partition information are spread again at the end to ensure latest information for runtime.

4.4 Optimization

The three combination rules used in optimization are each implemented using visitors.

4.5 Plan translation

Plan translation is done via `toSqlString()`-functions present in each operator with the use of placeholders. Each SQL representation of a operator contains placeholders for the operator output table and possibly inner (meaning children) operators and is a select from its data sources.

List of Listings 2: Data exchange SQL template

```
SELECT (<<OP0>>) UNION ALL (<<OP1>>)
```

For example the representational template of a data exchange operator can be examined in listing 2. The placeholders `<OP0>` and `<OP1>` are replaced with their specific children operator ids during runtime. The resulting placeholders in the SQL statement are then replaced again when combining the SQL statement of this operator with that of the children.

List of Listings 3: Equi join SQL template

```
SELECT <RESULT> FROM (<<OPX>>) AS <OPX> INNER JOIN (<<OPY>>) AS
    <OPY> ON <JATTX> = <JATTY>
```

A more complex example is presented in listing 3, that shows the SQL template of the equi join operator. It is more complex as its uses additional projection attributes (`<RESULT>`), aliases children operators (`AS <OP1>`) results and uses join conditions (`<JATTX> = <JATTY>`).

This SQL statements are assigned to operators in the generated query tracker plan. That plan is then handed off to deployment and execution.

5 Evaluation

This sections presents the results of the first benchmarks of XDB. XDB was tested on Java 1.6 with MySQL as underlying database system. This ran on a cluster with eight virtualized nodes, each consisting of a Intel Xeon X5650 with 2.67 GHz clock, 8GB of RAM and a 80 GB hard disk as secondary storage. The software stack was chosen as the following: Xen as a VM monitor, CentOS 6, Cloudera CDH 4.5 Free Edition including Hive, MySQL 5.6.10 (using InnoDB as the default storage engine and read committed as the default transaction isolation level) as well as Java 1.6.

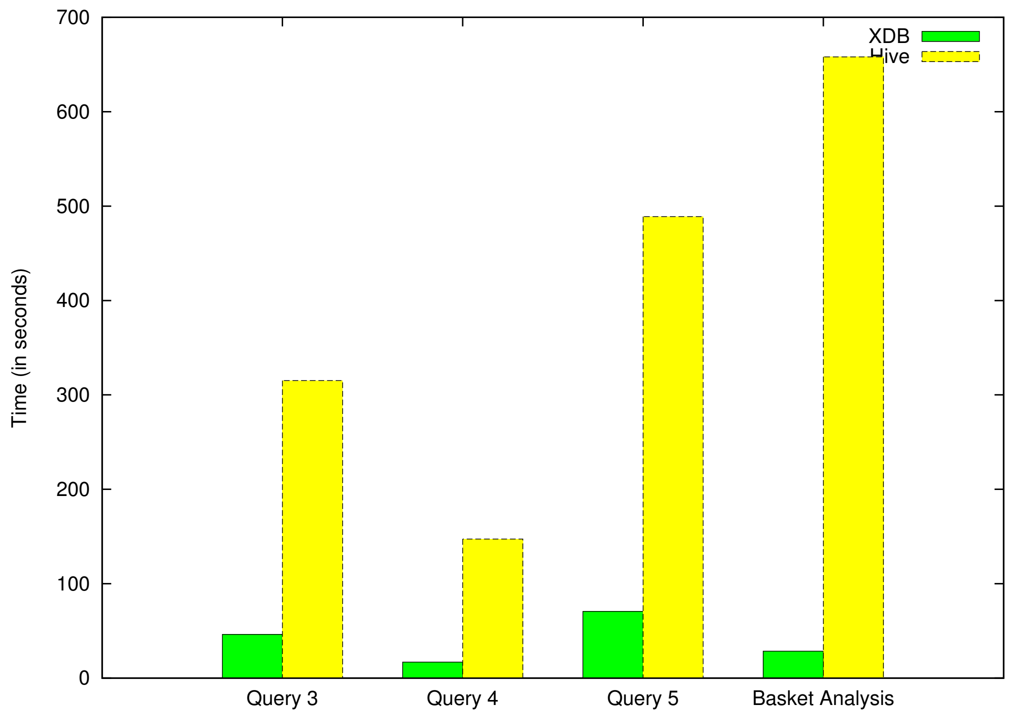


Figure 6: XDB vs. Hive (TPC-H SF=30) [3]

Tested were TPC-H queries Q3, Q4 and Q5 with a scaling factor $SF = 30$. Figure 6 shows the comparison between XDB and Hive executing the TPC-H benchmark on the same cluster. While Hive/Hadoop was installed and run with default configuration, XDB's table were partitioned: the *lineitem* table was partitioned into 8 parts using a hash-based partitioning scheme on the attribute *l_orderkey* and all other tables were partitioned using a reference based partitioning scheme such that all joins could be executed locally on each node with the partitioned *lineitem* table. As a result, a speed-up factor of approximately 6 to 10 in favour of XDB can be seen.

In order to show the flexibility of XDB, a basket analysis, which includes a white-box user-defined function as described in section 2.3 has been executed. For comparison, the basket analysis has been executed on Hive using the same Java code for computing the string-similarity in the user-defined function. The result of this experiment can be seen by the rightmost bar group of figure 6. Compared to Hive, XDB shows a speed-up factor of approximately 20 since XDB can optimize plans which include white-box user-defined functions (i.e., push down selections and aggregations over UDFs). [3]

6 Conclusion

6.1 Conclusion

XDB combines the advantages of the named systems: It offers the scalability, robustness and flexible programming model in a distributed environment like Hadoop does while using the stability and single query performance offered by relational databases. Meanwhile, compared to Hadoop/Hive, the performance could be increased by an amount that enables XDB systems to handle even the requirements of distributed big data applications. This paper showed that problems of other known systems could be solved and an implementation is feasible.

6.2 Outlook

Some implementational problems still exist, though. The current cost based analysis is only a very rough heuristic and should be replaced by a approximation based on real information about amounts of data handled by operators.

Furthermore more optimization rules could be implemented. Current borders for optimization rules are parallelized sub-plans, for which there already are approaches to overcome this problem.

References

[1] M. Abdelguerfi and K.-F. Wong. *Parallel Database Techniques*. IEEE Computer Society, 1998. ISBN 0818683988.

[2] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. FunSQL: it is time to make SQL functional. In *EDBT/ICDT Workshops*, pages 41–46, 2012.

[3] C. Binnig, A. C. Müller, S. Listing, T. Jacobs, C. Pinkel, S. Pfistner, L. Schmidt, and R. Sichler. XDB - An Elastic, Robust and Flexible Database Architecture for Big Data (Vision Paper). unpublished, 2013.

[4] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Commun. ACM*, 35, 1992.

[7] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[8] M. G. Noll. Running hadoop on ubuntu linux (single-node cluster). URL `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/`.

[9] I. Oracle. Using Replication for Scale-Out, May 2013. URL `https://dev.mysql.com/doc/refman/5.1/en/replication-solutions-scaleout.html`.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[11] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database systems: the complete book*. Prentice Hall Upper Saddle River, 2001.

[12] J. Venner. *Pro Hadoop*. Apress, 2009. ISBN 9781430219422.

[13] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

[14] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1060–1071. IEEE, 2010.

[15] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(5):611–636, 2012.