Distributed SQL Query Engine

Master-Level Semester Project Report

Amer Chamseddine and Artyom Stetsenko
{amer.chamseddine, artyom.stetsenko}@epfl.ch

Supervised by:
Professor Christoph Koch, Data Analysis Theory and Applications Laboratory

School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne

Lausanne, Autumn 2012

Abstract

Distributed relational databases are appealing when we want to store large data sets, since data is partitioned across several nodes that can each manage its own, smaller share of it. However, this approach comes with its own challenges. With partitioned data, it is difficult to know how, given an arbitrary query, it must be executed on the different nodes in order to produce the correct result. It is not possible to simply execute the query on each node individually and combine the results, since the query might be calling for data that is available on some nodes but not on the others. Aggregation and nested queries provide further challenges. Because of all this, careful planning and intelligent pre-shuffling of the data is required. It is not surprising then that there are no open-source relational distributed database systems capable of executing SQL queries with aggregates, as of the time of writing this report. This project is an attempt at building such a system.

Introduction

Large-scale data analytics has been the topic of intense commercial interest as well as academic research for many years. The growth of storage systems beyond a single machine have rendered traditional relational database management systems unusable for data storage for the purpose of easy querying. Moreover, the difficulty of running SQL queries in a distributed fashion has led to the relaxation on the requirements of data management and emergence of the NoSQL paradigm, where data is not stored in relations and where traditional SQL cannot be used for data manipulation. While it is true that NoSQL databases are highly optimized for retrieval and appending operations, they provide little functionality beyond that. The common approach to doing analytics on data stored in a NoSQL database is therefore to extract the necessary portions of the data and manually run them through a statistical package. It is needless to say that this approach is time-consuming and error-prone.

While plain SQL does not provide rich analytical capabilities itself, it does provide useful data manipulation techniques that can be applied to do at least some analytics in-database. Support for user-defined aggregate functions is also on the rise among modern RDBMSs, expanding the boundaries of analytics that can be performed by using SQL queries. Ability to create user-defined aggregate functions is readily supported by PostgreSQL[1], Oracle[2], and SQL Server[3], among possibly others, and PostgreSQL even has a full-fledged open-source data analytics library available, called MADlib[4]. Despite these advances in RDBMS capabilities, relational databases still struggle in growing beyond a single node. In this project, we attempt to build a distributed query engine that can execute any arbitrary SQL query on a database partitioned on a set of worker RDBMS nodes. Our focus is to use each worker node's capabilities to its fullest, doing as much

[1] 2012. PostgreSQL: Documentation: 9.2: User-defined Aggregates. http://www.postgresql.org/docs/9.2/static/xaggr.html.

[2] 2007. Using User-Defined Aggregate Functions. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28425/aggr_functions.htm.

[3] 2008. Create User-defined Aggregates. http://msdn.microsoft.com/en-us/library/ms190678.aspx.

[4] 2009. MADlib. http://madlib.net/.

processing in parallel as possible, while at the same time taking into account the fact that the query may contain aggregates and nested queries, and that the underlying data partitioning may be arbitrary. Our system is implemented as a Java application using PostgreSQL worker nodes.

In the Design section, we discuss how we go about executing an arbitrary query and describe our algorithms. In the Implementation section, we give the implementation details of our ideas and present our pipeline of query execution. In the Evaluation section, we give details on how well our system performs. We conclude in the Conclusion section and describe planned future extensions of this system in the Future Work section.

Design

We start off the discussion about our system by describing its design. To that end, we first present the architecture of our system. We then give an overview of distributed query execution and describe how it is different from the planning stage of any non-distributed query execution pipeline. We then state our major assumptions about our system, describe the major algorithms we came up with, and finally describe how we go about executing a given query.

System Architecture

The architecture we decided to use for our system is a simple two-tiered architecture where a single master node controls a set of worker nodes that each runs its own (single-node) RDBMS instance. The worker nodes are hidden from the user, who only interacts with the master, and to whom the master appears as a fully-functional RDBMS itself. The worker nodes, on the other hand, operate independently of each other and are not aware of each other, as all interaction goes through the master. This enables us to have an arbitrary number of worker nodes, as long as the master is aware of them.

In other words, our system can be viewed as a layer on top of a set of RDBMS instances. This architecture was inspired by Milestone 2 of our Advanced Databases course project from the Spring 2012 semester. The requirement there was to use a set of eight worker MySQL nodes and build a system that executes two queries on a database partitioned on these nodes. In fact, the implementation of the current system is based on our implementation of Milestone 2.

Major Assumptions

While building our system, we made several major assumptions on our system and on the partitioning scheme, which we would like to disclose next.

First of all, we assume that the more work we push down to our RDBMS worker nodes (rather than do in our code), the better performance we can achieve. We believe this is true mainly because of two reasons: (1) the data and operations on the data are kept in one place, and (2) the optimizer already available in the underlying RDBMS is used and need not be reinvented.

Second, we assume that the data is partitioned on the worker nodes uniformly and without overlap. That is we assume that no part of any relation is replicated, and that horizontal (and not vertical) partitioning is employed. We made this assumption in order to avoid having to handle duplicate tuples that may come from different worker nodes. However, we understand that replication can provide significant boost to query execution, and support for it is something that could be

incorporated in the future.

Finally, we assume that we achieve the best performance when each node produces its share of the query results given the underlying partitioning (we make sure the node has access to all the data it needs), rather than a small subset of nodes doing a lot more work. We believe this is true because each node would perform more or less the same amount of work, and thus the query execution would be maximally parallelized.

Overview of Distributed Query Execution

Any modern DBMS, when executing a query on a single node, typically performs the following three steps for a given query:

1. Parsing
2. Planning
3. Execution

In the parsing stage, a given query string is parsed and transformed into a formal data structure called a parse tree. This parse tree is then given to the query planner, which, using the system catalog that has information about each relation, decides on an efficient way to execute the query and builds another data structure called a query plan. This query plan is then handed off to the query executor which executes it.

Given our chosen system architecture model, the master node is the one that must ensure that the given query gets executed. To that end, it does produce a parse tree, but instead of producing a typical query plan reminiscent of regular RDBMS query plans, it produces a *distributed* query plan. In other words, it comes up with a set of steps that must be executed on our given worker nodes in order to produce the correct final result. The master then executes these steps by sending SQL queries to the worker nodes.

We would like to stress that distributed query planning is different from non-distributed query planning in the sense that the operators are not the same. Although our system does not do this at this stage of development, distributed query planning could also choose among several query plans, given how the underlying data is partitioned. While building our system, we tried to make it work for any underlying partitioning, and hence we do not take it into account. Query optimization was not our primary focus, but rather something that could be added in the future.

Aggregation

In our architecture model, if every worker node has all the data necessary to execute the query (given its partitioned share of the database), all we have to do is to execute the query on all nodes and then combine the results. This is true even if a query contains nested queries. However, aggregated queries pose a slight problem. Without nested queries, if our given query is an aggregate query (e.g. it contains an aggregate function such as SUM), we can still pass it to the workers, but combining the results would be more difficult. This is because, with some aggregate functions, the nodes need to return not the final result of the aggregate function, but rather its final *state*. This is easily illustrated by the AVG function, which maintains as its state a sum of elements and their count. If our query contains this aggregate, we need our nodes to return the final states of this aggregate so that we can combine them and produce the final aggregate value.

Note that the function producing the final aggregate state on each node and the function combining

the different aggregate states are also aggregate, but they are different from their regular, non-distributed variant of the aggregate function. We refer to the two functions as *intermediate* and *final* variants of a given aggregate function.

Queries containing nested aggregate queries pose a further problem. Because the result of the outer query depends on the result of the nested query, and because, as shown above, the result of any aggregate query must be computed as a combination of the results from all worker nodes, we have to execute the nested aggregate query first and notify the workers of its result before attempting to execute the outer query on them.

SuperDuper Algorithm

The fundamental algorithm of our system is called the SuperDuper algorithm. It is based on Bloom Join, a well-known algorithm for performing equijoins in distributed databases. Before we go into the description of our SuperDuper algorithm, we give a short reminder of Bloom Join.

Bloom Join

Bloom Join is an equijoin algorithm for joining two relations stored at different sites based on an equality between a field in each. Assuming relations R and S where S is the smaller relation, and an equijoin condition $R.a = S.b$, the algorithm works as follows:

On the site with R:

1. Choose a hash function $hash$ with range $[0; k-1]$, where k is an integer in the order of the cardinality of S.
2. Construct a vector of k bits and set each bit to 0.
3. For each tuple $r \in R$, compute $h = hash(r.a)$ and set the hth bit in the bit vector to 1.
4. Ship the final bit vector to the site with S and wait for results.
5. Upon receiving the result set, perform an equijoin locally.

On the site with S:

1. Receive the bit vector from the site with R
2. Construct an empty result set S^*.
3. For each tuple $s \in S$, compute $h = hash(s.b)$ using the same hash function $hash$. If the hth bit is set in the bit vector, append the tuple to S^*.
4. Ship S^* to the site with R.

The Bloom Join algorithm proves very efficient because the amount of data shipped is very small: a bit vector is cheap to ship, and S^* contains little useless data if a good hash function is chosen.

Node State Independency

To do the work along the lines of our third major assumption, where each node should produce its share of query results, we first have to ensure that each node is capable of doing so, i.e. it has access to all the data it needs. We say about a node that satisfies this criterion for a particular query that its state is *independent* of the states of the other nodes.

We next claim the following: the state of any node can be made independent of the states of other

nodes along any one equijoin condition. To see why this is the case, consider two relations R and S in a distributed database, and an equijoin condition $R.a = S.b$. We can assume that R and S are arbitrarily partitioned (horizontally and without overlap) on a set of nodes. It is easy to see that a node holding some partition $R' \subseteq R$ is able to produce its set of the equijoin results if it has all the tuples from S that match the equijoin (or vice versa), thereby making its state independent. By the same logic, the state of each other node can be made independent.

The Algorithm

Node state independency is the goal of the SuperDuper algorithm. Given a set of nodes, two relations R and S, and an equijoin condition $R.a = S.b$, it uses the Bloom Join algorithm to make the states of each node independent along the condition. This is achieved by running the Bloom Join algorithm between each pair of nodes, shipping the necessary portions $S^* \subseteq S$ to the locations of their matching $R' \subseteq R$. All steps of the Bloom Join algorithm are executed except the last (local equijoin), which is implicitly part of the query that will be executed on each node.

The name of the algorithm is derived from an earlier version of it where the last step of Bloom Join was not yet omitted. After running that version of the algorithm on relations R and S, we would jokingly refer to the resulting relation as "Super R" (or "Super S", depending on the direction of the algorithm). A further SuperDuper of Super R with a relation T would then produce a "Super Duper R", a reference that eventually contributed to the algorithm's name.

SQL Query As a Graph

Taking the idea of being able to execute equijoins in a distributed manner by using the SuperDuper algorithm, we can further express an SQL query as a graph of equijoins. In such a graph, each relation of the query would be represented by a vertex, with an edge between two vertices representing an equijoin condition. Vertices representing physical relations are themselves physical, whereas vertices representing nested relations (i.e. in the case of nested queries in the FROM clause) are represented by supervertices, which can in turn contain vertices and edges. These vertices are named after the relations they represent. In addition, we represent any aggregate (sub)query as an additional, anonymous supervertex. Because the graph may contain supervertices, we often refer to it as a supergraph. Moreover, because no vertex can belong to more than one supervertex, the graph is a simple supergraph.

While processing the query, we execute each supervertex until completion, i.e. we execute the query by converting each supervertex to a physical vertex. As explained before, we need to run aggregate queries until completion at all times. This is the reason aggregate queries are mapped to supervertices, even if they are nested in the WHERE clause where we normally would not create a supervertex.

Our query graph does not contain any other information about the query, such as the projected fields or operators used in the WHERE clause. Knowledge of these is not necessary to make node states independent, and the nodes will know what to do afterwards when we give them the actual query. Such a graph is helpful for running a series of SuperDuper instances, making the state of each node independent of the states of other nodes along *every* equijoin condition of the query.

Internally, we refer to the algorithm we use to convert an SQL parse tree to a graph as the QueryTackler algorithm because it "tackles" the (bulky) query to produce a more simplified

representation of it.

Let's consider a simple database on which we can show a couple of examples. This database contains three relations: S (tudents), C (ourses), and T (ook), where S contains student IDs and names, C contains course IDs and names, and T expresses the many-to-many relationship between students and courses by containing student IDs and course IDs as foreign keys, indicating that a specific student has taken a specific course. Without loss of generality, let's assume also that each of the three relations is partitioned arbitrarily (i.e. independently of each other) on a set of nodes.

Consider now a simple query:

```
SELECT S.name
FROM S
```

The graph of this query is very simple and looks as follows:

Figure 1. Graph of a simple query.

What this graph tells us is simply that there is nothing that needs to be done to make the state of each node independent of the states of the other nodes: since we assume non-overlapping partitions, $S' \subseteq S$ on each node is already independent of the others.

Now consider another query, returning all students who took the course with ID 5:

```
SELECT S.name
FROM S, T
WHERE S.id = T.sid AND
      T.cid = 5
```

The graph of this query is again quite simple and looks as follows:

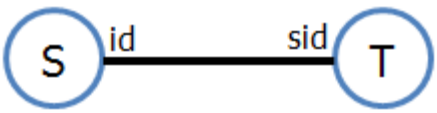


Figure 2. Graph of another simple query.

What this graph tells us is that we need to SuperDuper S to T (or vice versa) on the $S.id = T.sid$ condition to make the state of each node independent. After that, we can run the query on each node and combine the results.

Now consider a slightly more complicated query, returning all students who took the Advanced Databases course:

```
SELECT S.name
FROM S, T, C
WHERE S.id = T.sid AND
      C.id = T.cid
```

```
        C.name = 'Advanced Databases'
```

This query's graph looks like this:

Figure 3. Graph of a more complicated query.

We now need to perform two SuperDupers: S to T and C to T. Note that the direction of the first SuperDuper does not matter.

Finally, let's consider an even more complicated query, a modified version of Query 7 from the TPC-H benchmark[5] (the query has been modified to remove an OR operator):

```
SELECT shipping.supp_nation,
       shipping.cust_nation,
       shipping.l_year,
       SUM(shipping.volume) AS revenue
FROM (
      SELECT N1.n_name AS supp_nation,
             N2.n_name AS cust_nation,
             EXTRACT(YEAR FROM L.l_shipdate) AS l_year,
             L.l_extendedprice * (1 - L.l_discount) as volume
      FROM supplier S,
           lineitem L,
           orders O,
           customer C,
           nation N1,
           nation N2
      WHERE S.s_suppkey = L.l_suppkey AND
            O.o_orderkey = L.l_orderkey AND
            C.c_custkey = O.o_custkey AND
            S.s_nationkey = N1.n_nationkey AND
            C.c_nationkey = N2.n_nationkey AND
            N1.n_name = 'FRANCE' AND
            N2.n_name = 'GERMANY' AND
            L.l_shipdate BETWEEN '1995-01-01' AND '1996-12-31'
      ) shipping
GROUP BY shipping.supp_nation,
         shipping.cust_nation,
         shipping.l_year
ORDER BY shipping.supp_nation,
         shipping.cust_nation,
         shipping._year
```

[5] 2003. TPC-H - Homepage. http://www.tpc.org/tpch/.

While the query is obviously more complicated than the previous, its graph looks as follows:

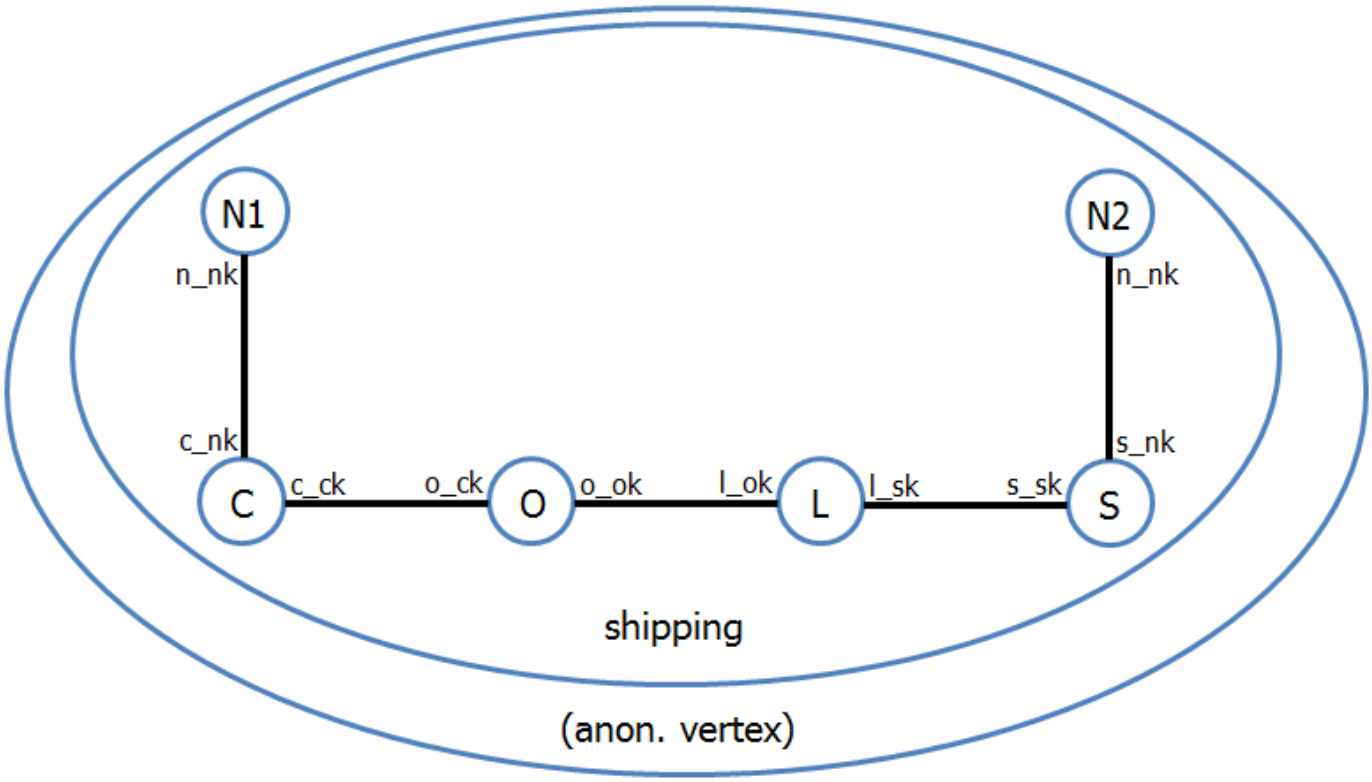


Figure 4. Graph of TPC-H Query 7.

Legend:

- nk = nationkey,
- ck = custkey,
- ok = orderkey,
- sk = suppkey.

As can be seen, this graph does not look as complex as the query itself.

Query Graph Processing and Execution of the Query

Our goal is to come up with a distributed query plan for the query at hand. For solving this problem we devised two further algorithms in our query execution pipeline: the GraphProcessor algorithm and the StepExecutor algorithm.

GraphProcessor Algorithm

The GraphProcessor algorithm takes as input the query graph, as produced by the previous step in the pipeline, and produces as output a sequence of elementary *execution steps*, to be executed later by the StepExecutor algorithm. An execution step can be one of three types:

- SuperDuper: performs a SuperDuper given an equijoin condition between two relations.
- Gather: gathers a table partitioned on all nodes into a non-distributed table on the master node.
- Run-Subquery: executes an actual (sub)query, either on all nodes or only on the master node.

With the above three constructs, the GraphProcessor algorithm produces the distributed query plan. To build the sequence of elementary steps, the GraphProcessor algorithm "consumes" the query graph. Its goal is to transform the graph into a single physical vertex, because if we have a graph consisting of a single physical vertex, then we already have the result of the query stored in the table corresponding to this vertex.

The GraphProcessor algorithm can process any graph. It does so step by step, at each step producing an action that should be performed in order to transform the graph from its old state to a new, simpler state. This action is nothing but one of the execution steps described earlier; it gets enqueued as a step in the sequence of steps constituting the output of the algorithm.

The GraphProcessor algorithm introduces a further distinction to the physical vertices of the graph. They can be of two types: distributed (D), i.e. those distributed (partitioned) among all worker nodes, or non-distributed (ND), i.e. those fully residing on the master node.

The GraphProcessor algorithm acts on the graph recursively. As mentioned in the previous section, our graph is a simple supergraph: a vertex cannot belong to two different supervertices. We can use this property to recursively traverse the graph, at every level performing three main operations:

- transform supervertices into physical vertices;
- "eat" all edges connecting any two physical vertices;
- generate the sequence of execution steps corresponding to the above transformations.

The above operations are achieved by a method called *process*. The method takes as input the super vertex and returns the physical vertex that should replace it in the graph. For every supervertex sv in the graph, the method transforms all sv's supervertex children into physical vertices by *process*ing them, after which we are left with a normal graph containing only physical vertices and edges. The next step consists of "eating" the edges. Since edges represent equijoin conditions, they are "eaten" by a SuperDuper step, which the algorithm adds to the sequence of execution steps. After this, the vertices attached to one end of the eaten edge get fused into the vertex on the other end. When there are no more edges, the graph consists of one or more physical vertices.

If the current graph has more than one vertex (i.e. the original graph was not fully connected), then a cross join needs to be performed between the vertices. The algorithm currently does not support performing cross joins in a distributed manner, so it will issue commands (enqueue execution steps) to ship all concerned relations to the master node and run the corresponding subquery there.

On the other hand, if the graph has only one vertex (i.e. the original graph was fully connected), then the algorithm runs the subquery on the data in place, and only gathers the result on the master node if necessary, e.g. if the subquery is aggregate. Four cases apply here (in the below "the vertex" refers to the single remaining vertex, and "the subquery" refers to the subquery attached to the super vertex we are currently processing):

- the vertex is non-distributed and the subquery is not aggregate;
- the vertex is non-distributed and the subquery is aggregate;
- the vertex is distributed and the subquery is not aggregate.
- the vertex is distributed and the subquery is aggregate.

In the first two cases where the vertex is non-distributed, we enqueue an execution step to run the subquery (being aggregate or not) directly on the master node. In the case where the vertex is distributed and the subquery in not aggregate, we just run the subquery on the worker nodes, and gather the result only if the current supervertex is the final one (the outermost in the graph).

In the last case, we need to execute the subquery on the worker nodes first, and then combine the results on the master, where the aggregation needs to be performed again. Hence, the query run on the worker nodes must output the final *state* of the aggregates rather than the final value, as explained before, and the query run on the master after gathering workers' results must be able to combine the different states. We refer to these first query as the *intermediate* query and to the second as the *final* query. We enqueue three execution steps: one to run the intermediate query on the workers, then one to gather the result on the master node, and the third to run the final query on the master.

The query plan generated for the given query graph is given to the StepExecutor for execution.

StepExecutor Algorithm

The StepExecutor algorithm executes the steps (which can be of the three types mentioned earlier) in order. After successful execution, the result is found in a table on the master node, which corresponds to the non-distributed physical vertex that remained in the graph after it was completely processed.

Implementation

This section describes our implementation details of the algorithms presented in the Design section, giving the details on how we communicate with our nodes and an overview of the whole query execution pipeline in order.

We implemented our system as a Java application using PostgreSQL worker nodes. We decided to use Java because we already had some code that we had written for our Advanced Databases course project. We chose PostgreSQL because it is an open-source DBMS which we eventually plan to integrate our system into.

Package Structure

Our code can be found in the subpackages of `ch.epfl.data.distribdb`.

- `ch.epfl.data.distribdb.lowlevel` contains all the low-level classes that we took from our Advanced Databases course project. This package contains the `DatabaseManager` class, which is responsible for sending queries to the various nodes and receiving data back from them.
- `ch.epfl.data.distribdb.parsing` contains all the logic for parsing of SQL queries.
- `ch.epfl.data.distribdb.tackling` consists of classes used to build the query graph.
- `ch.epfl.data.distribdb.execution` contains the logic for building and executing a distributed query plan for a query graph.
- `ch.epfl.data.distribdb.app` contains classes responsible for the user interface of our front-end application(s).

Database Manager

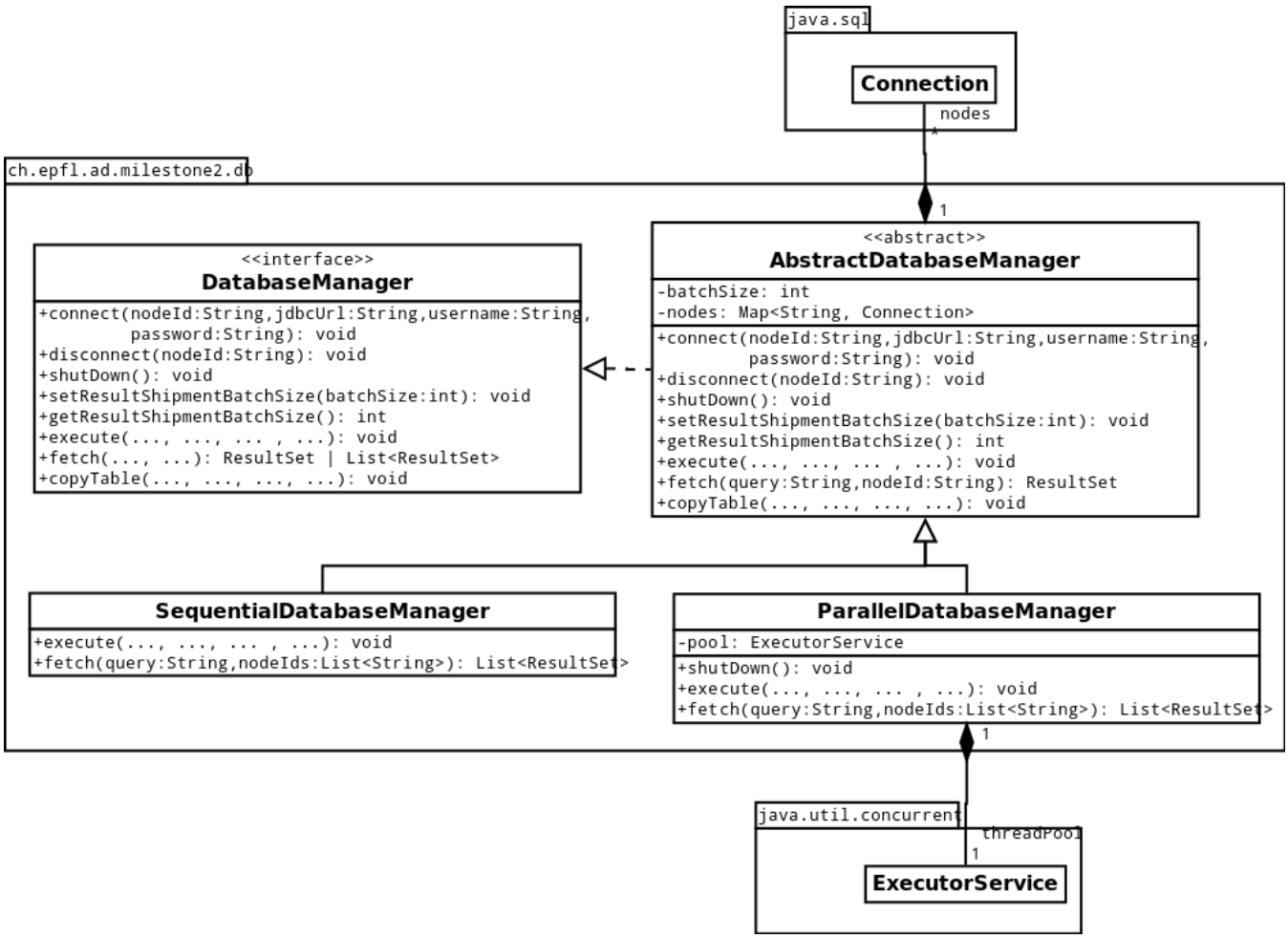
The Database Manager is a unified proxy for all JDBC accesses from our application. All our interaction with the master and the workers goes through its API instead of making direct JDBC calls. Therefore, it is a clean abstraction of all our nodes such that they appear as one simple entity to higher-level modules. With this abstraction, we could focus on the main logic of our system without worrying much about the details of JDBC and parallelisation across multiple nodes.

The design also allows different implementations when multiple nodes are involved: sequential versus parallel executions. This characteristic was extremely helpful during the development phase, as our program could be tested using the sequential implementation first (presumably simpler to realize) before moving to the parallel one without changes to the application code.

The versatility of this module comes from the plethora of API "flavours" for *execute*, *fetch*, and *copyTable* functions. The basic differences among these flavours lie in four aspects:

- Number of queries: one/multiple.
- Number of source nodes (where the queries are executed): one/multiple.
- Name and schema of the result table: one/multiple.
- Number of destination nodes (where results are stored): one/multiple.

Note that there is always some intrinsic logic involved in every flavour of the API. This is necessary so that code repetitions can be avoided in higher-level modules and most work can be pushed to database server nodes.



As seen from the UML class diagram above, the `AbstractDatabaseManager` class provides a basic implementation for `DatabaseManager` interface which defines the API. All generic JDBC operations, common configurations, as well as API methods agnostic to sequential or parallel execution are implemented here.

`SequentialDatabaseManager` extends `AbstractDatabaseManager` and implements the rest of the API in a naïve but straightforward manner: using for-loops when multiple inputs for some parameter are encountered. `ParallelDatabaseManager` extends `AbstractDatabaseManager` and implements the rest of the API in a parallel fashion using Java's built-in thread pool facility. The transition from `SequentialDatabaseManager` to `ParallelDatabaseManager` is intuitive: all for-loops are converted to constructions, submissions and invocations of tasks on the thread pools (the process analogous to the map function in Monad algebra). The calling thread awaits the completion of all tasks before proceeding to subsequent steps. Java's built-in `ExecutorService` provides a neat way to realise all these thread pool features.

Query Execution Pipeline

We execute a given query by following the following five steps in order, where the output of one is the input of the next:

1. Parsing (output: `QueryRelation`)
2. Graph construction (output: `QueryGraph`)
3. Graph processing (output: `List<ExecStep>`)
4. Step execution (output: `ResultSet`)
5. Printing of results and cleanup

Parsing

We use the General SQL Parser[6] library to parse a query string, and we convert the query tree produced by the parser to our own parse tree. The library seems accurate with two more or less significant drawbacks:

- The parsing error messages produced by the parser are not very informative.
- The parser does not perform sanity checks on the queries (e.g. allows aggregate fields to appear in GROUP BY clause).

Because we use a trial version of the library, it is limited to parsing only 10,000 characters. However, this limit does not seem to pose problems for our purposes.

The `Parser` class is responsible for parsing the query strings, and produces as output a parse tree represented by the `QueryRelation` class. The `QueryRelation` object is composed of several other objects representing the different aspects of the query:

- Projected fields, represented by a list of `Field` objects (an abstract class containing several flavors of extensions, such as `NamedField`, `AggregateField`, `LiteralField`, etc.).
- Relations, represented by the `NamedRelation` (physical relation) and `QueryRelation` classes derived from the abstract `Relation` class.

[6] 2005. General SQL Parser. http://www.sqlparser.com/.

- WHERE conditions (connected by conjunction), represented by a list of `Qualifier` objects, where each contains a reference to an `Operator` of the condition and a list of `Operand` objects (interface implemented by the `Field` and `Relation` classes)
- GROUP BY fields, represented by a list of `Field` objects.
- HAVING conditions, represented by another list of `Qualifier` objects.
- ORDER BY fields, represented by a list of `OrderingItem` objects, where each references a `Field` and an `OrderingType` (an enum that can take the values of `ASC` or `DESC`)

Graph Construction

The `QueryGraph` class represents a query graph. It takes in its constructor a `QueryRelation` object and constructs the corresponding graph. The graph construction takes place in two steps: first all vertices are recursively extracted from the parse tree, creating a set of `QueryVertex` objects, and then edges are recursively built, adding a set of `QueryEdge` objects to each vertex. Each vertex of the graph has access to its own query string, so that each vertex can be executed until completion, as explained before.

Edges that we build are directed. Edge direction was initially intended to be used to indicate nesting in correlated queries, such that the direction implies which of the two vertices is nested and which is not. Vertices at the same level of the query would have bi-directional edges. Since we do not support correlated queries at this point, all our edges are bi-directional.

Currently we only build edges when we encounter equality conditions in the WHERE clause between two fields belonging to two different relations (i.e. equijoin conditions). Theoretically, fields connected by the IN operator can also be considered as equijoin conditions, but currently we do not do that for two reasons:

- not building an edge does not produce incorrect results, and
- any non-correlated query using the IN operator can be rewritten such that the IN operator's subquery is moved to the outer query's FROM clause as an additional nested relation.

Graph Processing

The `GraphProcessor` class contains the logic that comes up with a distributed query plan for a query. It takes as input the query graph that was generated for this query, and processes the graph using heuristics for optimization. The output is a sequence of execution steps that are fed to the step executor in order to execute them.

The execution steps are represented by the `ExecStep` abstract class, from which the following three classes derive, each corresponding to the three execution steps introduced previously:

- `StepSuperDuper` represents the elementary step that consists in shipping tuples around, between worker nodes, according to the semantics described previously. For this, we keep track of the `fromRelation` and `fromColumn`, `toRelation` and `toColumn`, and `outRelation`. We also store a boolean `distributeOnly` that is used to distinguish the full SuperDuper step (where we ship from all worker nodes to all worker nodes) from the distribute-only version of SuperDuper where the `fromRelation` is gathered on the master node.
- `StepGather` represents the elementary step that gathers a distributed table on the master node. For this purpose, we keep a handle to the `fromRelation` (name of the

distributed table to be gathered) and `outRelation` (name of the new gathered table on master).

- `StepRunSubquery` represents the elementary execution step responsible for executing a (sub)query, either on all worker nodes or on the master. Besides storing the place where to execute the query in `stepPlace`, we also keep track of the string representation of the actual subquery in `query`, and the name of the relation to hold the results in `outRelation`. We also keep a flag that indicates whether the query is aggregate or not; but this is currently only useful for debugging purposes.

The implementation of `GraphProcessor` follows the design described previously. The key method is `process`; it contains the main logic. It takes a supervertex and returns the physical vertex that should replace it after having recursively processed it. In parallel with processing each vertex, it also stores the corresponding steps in the `execSteps` member variable.

This class contains helper functions too. `pickConnectedPhysical` and `pickConnectedND`, and `eatAllEdgesPhysical` and `eatAllEdgesND` are all used to pick a connected (D)istributed or (N)on-(D)istributed vertex as the destination of a SuperDuper step, and to eat all edges (and generated corresponding SuperDuper steps) between two (D)istributed or (N)on-(D)istributed vertices respectively. Initially we thought of always picking the largest relation as the destination of a SuperDuper step, in order to minimize the number of shipped tuples. Currently this is not implemented (we pick the first relation), but should be easy to implement given the current state: the two picker functions should simply be modified to check the size of the tables on the fly.

Step Execution

The `StepExecutor` class is used to execute the distributed query plan generated by the `GraphProcessor` class. It takes as input (using the `executeSteps` method) the list of execution steps constituting the plan to be executed. This method performs the execution according to the logic described in the Design section of this document, and computes the final result, and returns it as a `ResultSet`. The implementation is as simple as a for-loop that traverses the list of execution steps. For each one, according to its type, the class extracts the relevant fields, and either issues requests directly to the Database Manager or instantiates lower-level constructs such as `SuperDuper`.

SuperDuper

The `SuperDuper` class performs a SuperDuper operation on two relations. It sends tuples from the table `fromRelation` which is distributed on `fromNodeIds` to `toNodeIds`, such that a tuple is sent to node i if it can join with a tuple in the chunk of table `toRelation` on node i.

This is achieved by computing bloom filters on each `toRelation` chunk on all nodes in `toNodesIds` and sending them to all nodes in `fromNodeIds`, then filtering the `fromRelation` chunks with every received bloom filter and sending each result to the corresponding node. The join condition is an equijoin on the `fromColumn` of `fromRelation` and the `toColumn` of `toRelation`. The shipped tuples are stored in a temporary table called `outRelation`, on all nodes of the `toNodeIds` list.

The `SuperDuper` class uses a pool of threads to parallelize the job. The `shutDown` method should be called after the operation finishes, in order to shut the pool down. Failing to call this function

may result in the main program hanging (not exiting).

Printing of Results and Cleanup

We have two dedicated classes to manage the printing and cleanup of temporary tables.

The `TablePrinter` class contains a simple static function `printTableData` that is able to print a `ResultSet` to the console output in a nicely-formatted way. It automatically takes care of paginating the output and smartly adjusts the column widths according to the data displayed in the current page.

`TableManager` is the class responsible for allocating names for temporary tables, keeping track of who is using which temporary table, and cleaning up after the execution terminates. Higher-level classes such as `GraphProcessor` and `SuperDuper` use this class to request new table names using the `generateTmpTblName` method. The class makes sure that there is no duplication in the names, and keeps track of the requested names in the `tempTblNames` list, so that after finishing the execution the application (in our case the `CommandLine` class) calls `cleanTempTables` to delete all the temporary tables used as intermediary holders to run the specified queries.

Limitations

Our current system only supports SELECT queries. Data has to be loaded onto the worker nodes and partitioned separately before our system can be used. Ideally, our system would support all features offered by a standard PostgreSQL node, which is the end goal of this project. At this stage of its development, however, we focused only on the basic functionality, which is no doubt a building block towards the ideal system.

Because of limited time, several SELECT query constructs were also dropped from our implementation, as outlined below. Support for these is expected to be added in the future.

Correlated Subqueries

Our system was initially aimed at supporting correlated subqueries, and was designed with this support in mind. In fact, we were sure we supported them up until the last few days of the implementation, when we were applying finishing touches and trying the system out. Our initial idea was that correlated subqueries do not pose a significant problem; we were sure that we can ignore correlation altogether when building a query graph. That turned out to not be the case.

The query that proved our QueryTackler algorithm incompatible with correlated subqueries was this query using the Student-Took-Course schema introduced previously, which returns the students who took all courses:

```
SELECT S.name
FROM S
WHERE NOT EXISTS (
                  SELECT C.id /* any field */
                  FROM C
                  WHERE NOT EXISTS (
                                    SELECT T.sid /* any field */
                                    FROM T
```

```
                                    WHERE T.sid = S.id AND
                                          T.cid = C.id
                                )
                    )
```

For this query, our current algorithm produces the same graph as in Figure 3. However, SuperDupering S to T and then C to T is not enough to properly execute this query. What is needed instead is SuperDupering T to S and then gathering and distributing C to all nodes.

Because we discovered this bug late, we did not have time to come up with a generalization of how to execute such queries, and had to drop support for correlated queries altogether. However, we conjure that we can execute some correlated queries if we introduce a priority field to our graph vertices (indicating the level of nesting), forcing the GraphProcessor algorithm to pick lower-priority vertices first. Because we could not find any counter-examples, we believe that this extension would enable us to execute correlated queries that use EXISTS and IN operators, but not their negations.

Query Syntax

Our current implementation lacks support for the following elements of query syntax:

- * fields: because our parse tree structure needs to know all fields and because we cannot expand * fields, we currently do not support them (except the special case of COUNT(*)).
- JOIN clauses: our system currently only supports inner joins, and they should be expressed via the WHERE clause.
- OR and NOT operators in qualifiers: our qualifiers (both in WHERE and in HAVING clauses) are currently connected by conjunctions; theoretically it should be possible to implement OR and NOT as qualifier Operators, in which case the Qualifier class would have to implement the Operand interface, but that would introduce recursion.
- IN lists.
- IF and CASE clauses.
- Compound queries (UNION, MINUS, etc.).
- Other esoteric syntax and vendor-specific SQL extensions.

Evaluation

For evaluating our system, we used the TPC-H schema introduced above, where the relations Customer, Orders, and LineItem were partitioned equally on all nodes, and the other relations were stored fully on one node. Our system, however, does not know how the data is partitioned so the execution is the same as it would be if all relations were partitioned arbitrarily. We evaluated our system on scales 0.001 and 0.01 of the TPC-H benchmark database, on all queries of the benchmark that are currently supported by our system (12 out of 22).

Because we did not have access to a networked set of PostgreSQL nodes, we evaluated our system on one PostgreSQL server with 8 databases, where each stored a partition of the data. The machine is a Toshiba Tecra laptop with an Intel(R) Core(TM)2 Duo CPU T9400 running at 2.53GHz, and 3.8 GiB of RAM. We used PostgreSQL 9.0.

Query	0.001	0.01
1	1.544 s	0.928 s
3	7.034 s	12.8 s
5	14.795 s	18.825 s
6	0.818 s	0.326 s
7	15.989 s	35.661 s
8	17.593 s	39.225 s
9	15.47 s	31.959 s
10	9.813 s	34.362 s
11	0.772 s	1.039 s
13	3.016 s	6.161 s
14	2.567 s	6.91 s
18	1.988 s	7.151 s

Table 1. Evaluation results.

As can be seen in the table, for some queries the execution time on the smaller dataset is higher than the execution time on the larger dataset. This is the case because we have a randomness factor in the code: the `GrpahProcessor` can generate different distributed query plans for the same query, due to the way we currently pick a connected vertex to be the destination of the SuperDuper step. As mentioned earlier, we currently pick the first vertex we find (and it is random because of the properties of `HashSet` in Java), but in the future the picker functions should pick more intelligently (e.g. according to a specific heuristic), in which case the distributed query plan will be deterministic.

Conclusion

We have presented a system consisting of a layer that interposes between the end user and an array of commodity PostgreSQL nodes and provides an interface for running any arbitrary query in a distributed fashion[7]. The system efficiently processes complex queries by scaling their execution onto tens of nodes. It comes up with an optimized distributed query plan for the given query, which optimizes the amount of data shipped around. It also pushes as many tasks as possible to the worker nodes in order to parallelize the query execution. All in all, the system is a highly efficient distributed query engine that scales polynomially with the number of nodes.

Future Work

We do not consider the work to be finished. Besides fixing the limitations mentioned in the

[7] Modulo the limitations mentioned in the Limitations section.

Limitations section, there are quite a few things that can be added or optimized in our system. We list the major ones below.

Extending the Query Graph

The query graphs that we build currently are basic. They do not encode any information about the query other than the equijoin conditions. While such conditions are definitely used the most in everyday database querying, there are other conditions that are also used frequently that we currently do not take into account.

The simplest of such conditions are unary (i.e. pertaining to only one relation) qualifier conditions in the WHERE clause. They can be used to limit enormously the amount of data we ship around, and are not that hard to add to our current system. We were envisioning attaching them to our graph vertices, such that the GraphProcessor algorithm can take advantage of them.

Moreover, there are of course all of the relation joins that are not equijoins. These, however, would be more challenging to handle.

Integration with MADlib

Our existing system provides a clean way to execute aggregate functions on the database. All that needs to be done is that the non-distributed variants of these functions need to be split into two each, the intermediate and the final flavors, as mentioned above. MADlib provides a plethora of aggregate functions and includes all the means to provide the two flavors (i.e. the *merge* function that can merge two different states of an aggregate into one state, which is used as the state transition function for the final version of the aggregate function). For this reason, we believe our current system can be easily extended to support MADlib.

Integration with PostgreSQL

Many limitations of our system stem from the fact that it operates the master node remotely. This also results in below-optimal performance. An excellent continuation of this project would be to port it to C and integrate it into PostgreSQL, namely the master node. This is advantageous for many reasons, some of the most important of which are listed below:

- the master node will gain access to the catalog of the database, and will be able to make more efficient decisions about graph processing;
- the system will be able to make use of built-in parser of PostgreSQL;
- the system will be "closer to the data" and most likely closer to the worker nodes as well;
- clients currently using a simple non-distributed version of PostgreSQL will be able to seamlessly move to our distributed version of the system.