Evaluation Model

This section describes how a query is evaluated in the database. In ClustrixDB we slice the data across nodes and the query is sent to the data. This is one of the fundamental principles of the database that allows it to scale almost linearly as more nodes are added.

For concepts about how the data is distributed, please refer to Data Distribution as this page will assume an understanding of those concepts. The primary concepts to remember are that tables and indices are split across nodes, and that each of them has its own distribution that allows us to know, given the lead column(s), precisely where the data lives.

Parallel Query Evaluation (by example)

ClustrixDB uses parallel query evaluation for simple queries and Massively Parallel Processing (MPP) for analytic queries (akin to columnar stores).

It's best to take a look at some examples to understand query evaluation and why queries scale (almost) linearly with ClustrixDB. Let's start with a SQL schema and work through some examples.

```
sql> CREATE TABLE bundler (
     id         INT      default NULL auto_increment,
     name       char(60) default NULL,
     PRIMARY KEY (id)
);

sql> CREATE TABLE donation (
     id         INT default NULL auto_increment,
     bundler_id INT,
     amount     DOUBLE,
     PRIMARY KEY (id),
     KEY bundler_key (bundler_id, amount)
);
```

Now, let's see what the data distribution looks like in this case. We have three representations:

- _id_primary_bundler
- _id_primary_donation
- _bundler_key_donation

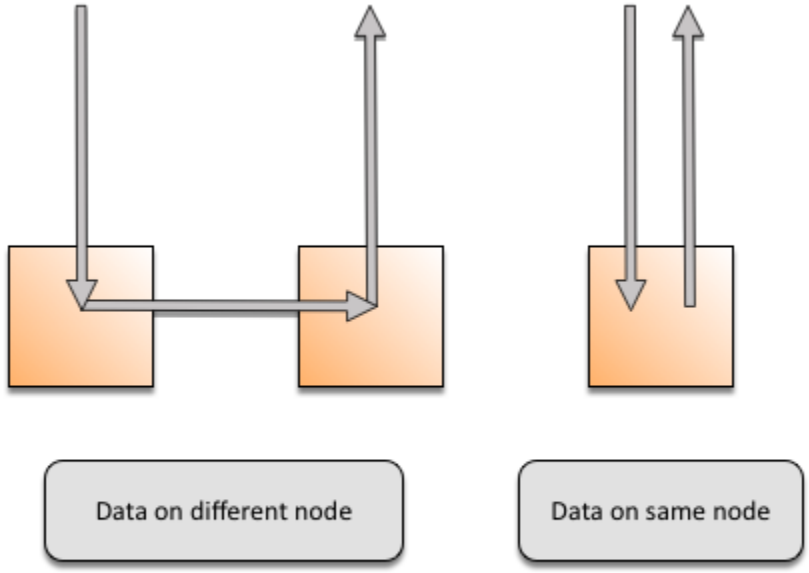
The first two are distributed based on their id fields. The last is distributed based on bundler_id. Please note that since these are hash distributed, all rows with same key value go to the same node.

Scaling Simple Queries

Here, we define simple queries as point selects and inserts. Let's consider a simple read (a simple write follows the same path):

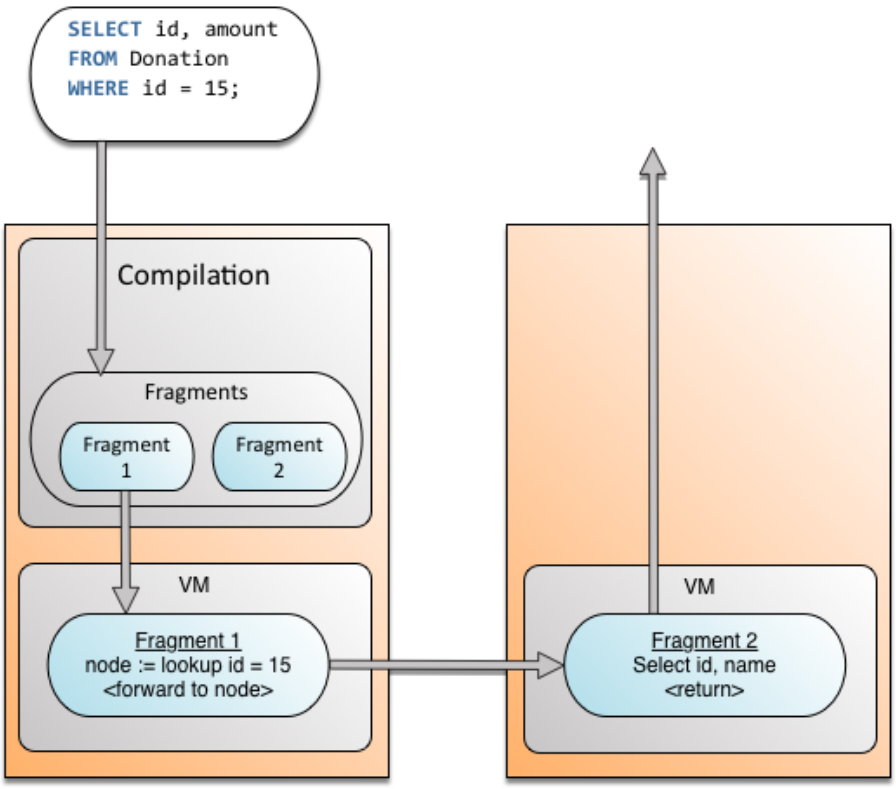
```
sql> SELECT id, amount FROM donation WHERE id = 15;
```

The data will be read from the ranking replica. This can either reside on the same node or require one hop. The diagram below shows both cases. As dataset size and the number of nodes increase, the number of hops that one query requires (0 or 1) does not change. This allows linear scalability of reads and writes. Also, if there are two or more replicas (which is normally the case) there will be at least one hop for writes (since both replicas cannot reside on the same node).



Query Compilation to Fragments

Now, let's dig a little bit deeper to understand how query fragmentation works. Queries are broken down during compilation into fragments that are analogous to a collection of functions. For additional information, please see Query Planner and Optimizer. Here we'll focus on the logical model. Below, you can see the simple query compiled into two functions. The first function looks up where the value resides and the second function reads the correct value from the container on that node and slice and returns it to the user (the details of concurrency etc. have been left out for clarity).



Scaling Joins

Joins require more data movement. However, ClustrixDB is able to achieve minimal data movement because:

- Each representation (table or index) has its own distribution, therefore we look up which node/slice to go to next, removing broadcasts.
- There is no central node orchestrating data motion. Data moves to the node needing it next. This reduces hops to the minimum possible, given the data distribution.

Let's look at a query that gets the name and amount for all donations collected by the particular bundler, known by id = 15.

```
sql> SELECT name, amount from bundler b JOIN donation d on b.id = d.bundler_id WHERE b.id = 15;
```

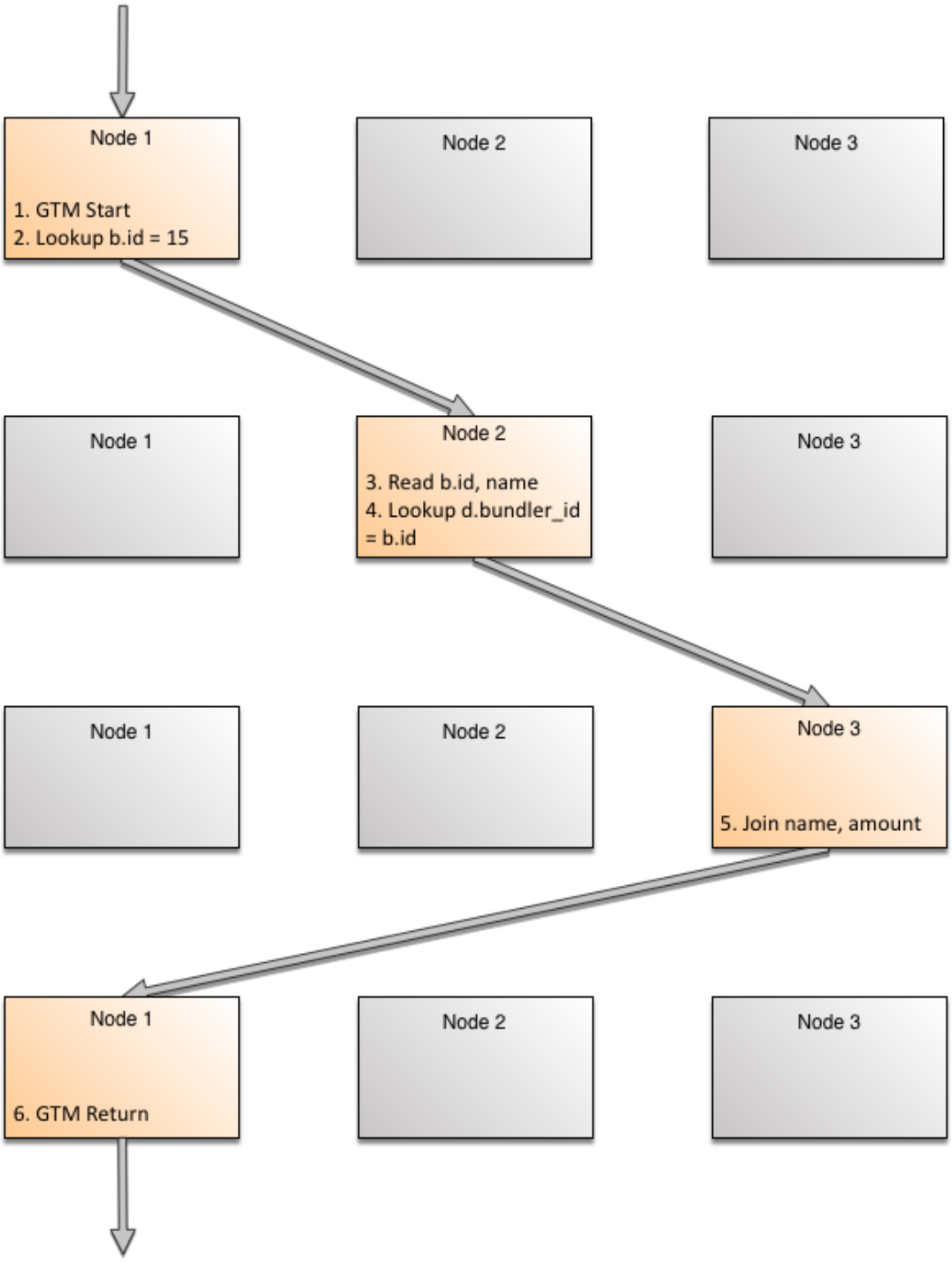
The SQL optimizer will look at the relevant statistics including quantiles, hot lists, etc. to determine a plan. The concurrency control for the query is managed by the starting node (that has the session), called the GTM (Global Transaction Manager) node for that query. It coordinates with Local Transaction Manager running on each node. For details see Concurrency Control. The chosen plan has the following logical steps:

1. Start on GTM node
2. Lookup nodes/slices where _id_primary_bundler has (b.id = 15) and *forward* to that node
3. Read b.id, name from the representation _id_primary_bundler
4. Lookup nodes/slice where _bundler_key_donation has (d.bundler_id = b.id = 15) and *forward* to that node
5. Join the rows and *foward* to GTM node
6. Return to user from GTM node

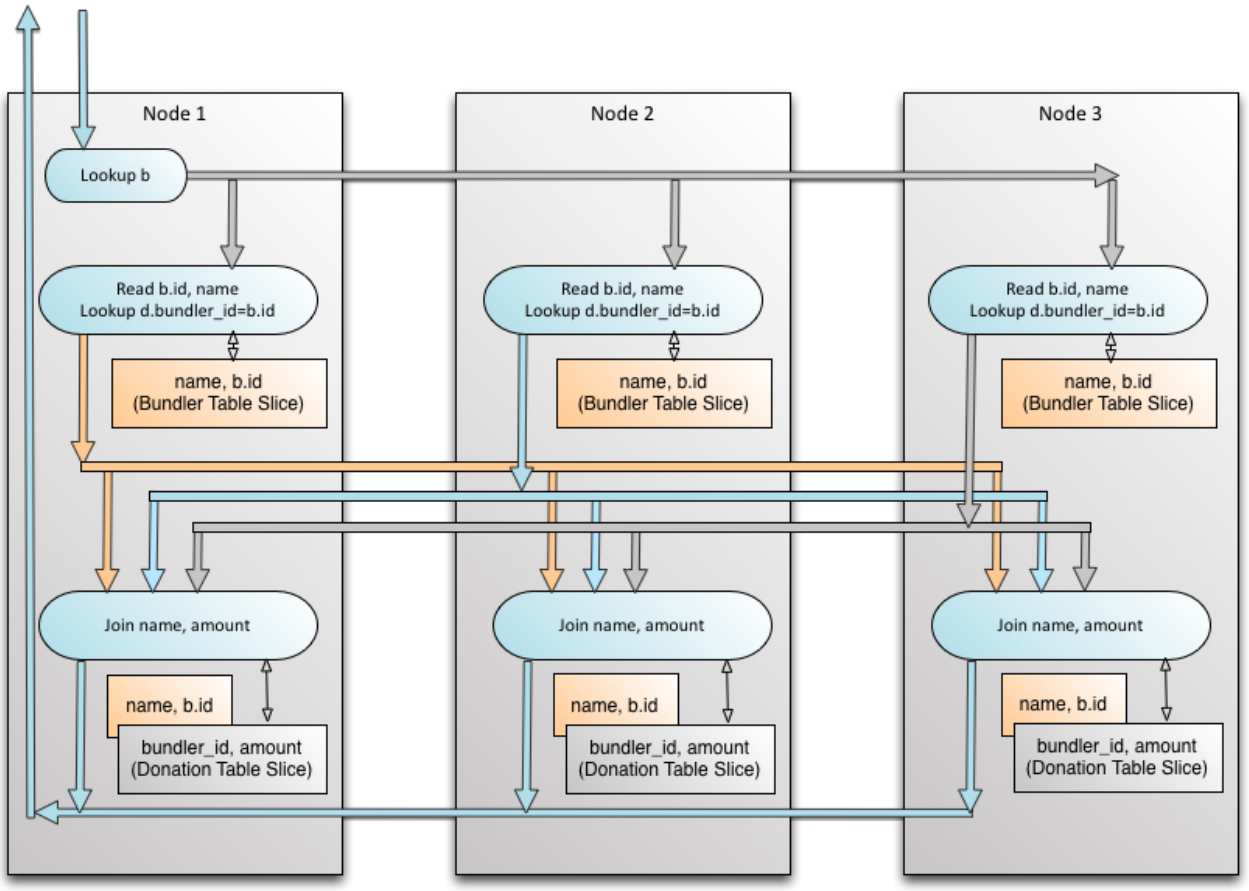
The key here is in Step #4. For joins, when the first row is read, there is a specific value for the join column. Using this specific value, the next node (which might be itself) can be looked up precisely.

Let's see this graphically. Each returned row has gone through the maximum of three hops inside the system. As the number of returned rows increases, the work per row does not. Rows being processed across nodes and rows on the same node running different fragments, use concurrent parallelism by using multiple cores. The rows on the same node and same fragment use pipeline parallelism between them.

We'll assume we have three nodes. For clarity, the three nodes are drawn multiple times to depict every stage of query evaluation.

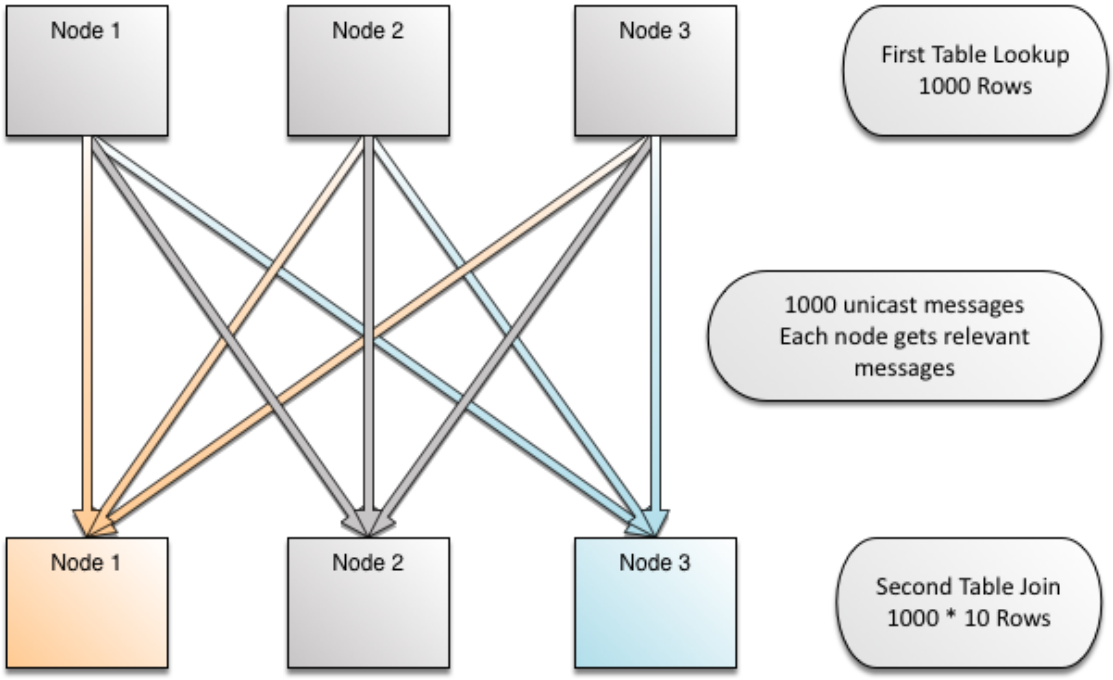


Now that we have seen the path of a single row, let's assume we didn't have the b.id=15 condition. The join will now run in parallel on multiple machines since b.id is present on multiple nodes. Let's look at another view, this time the nodes are drawn once and the steps flow downward in chronological order. Please note that while the forwarding for join shows that rows may go to any node, this is in context of the entire sets of rows. One single row is usually only forwarded to one node.



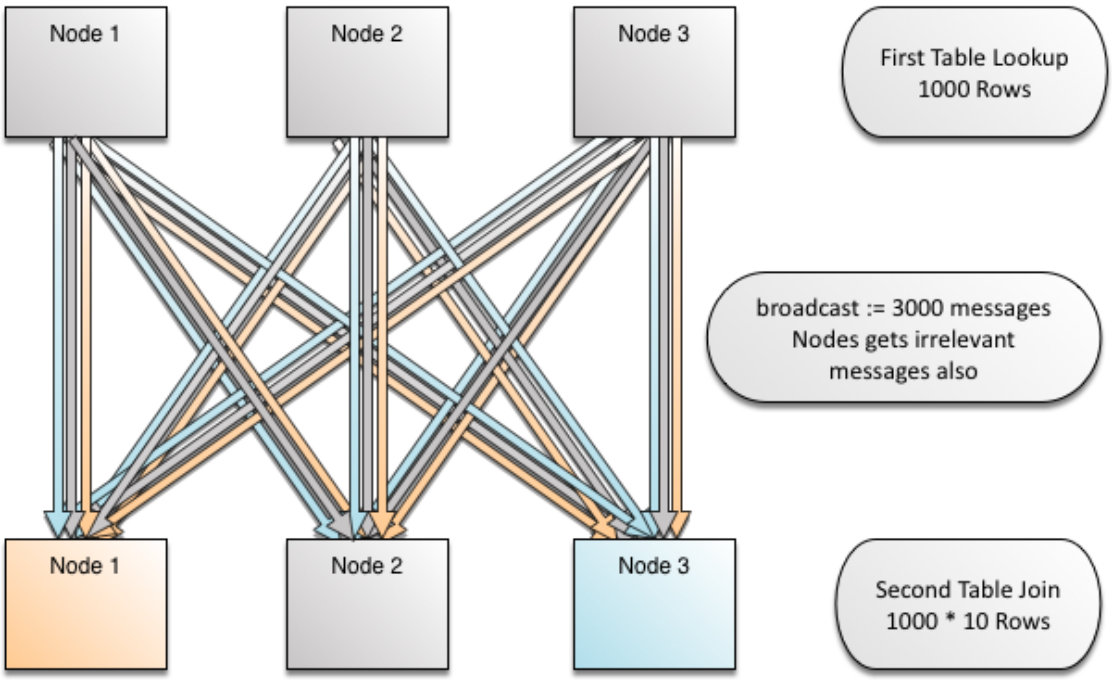
Joins with Massively Parallel Processing (ClustrixDB)

For a single row, we can see how ClustrixDB is able to precisely forward it by using a unicast. Now, let's zoom out and see what a large join means at a system level, and how many rows are forwarded in a cluster. Each node is getting only the rows that it needs. The colors on the receiving nodes match those on the arrows - this is to indicate that rows are flowing correctly to nodes where they will find matching data for successful join.



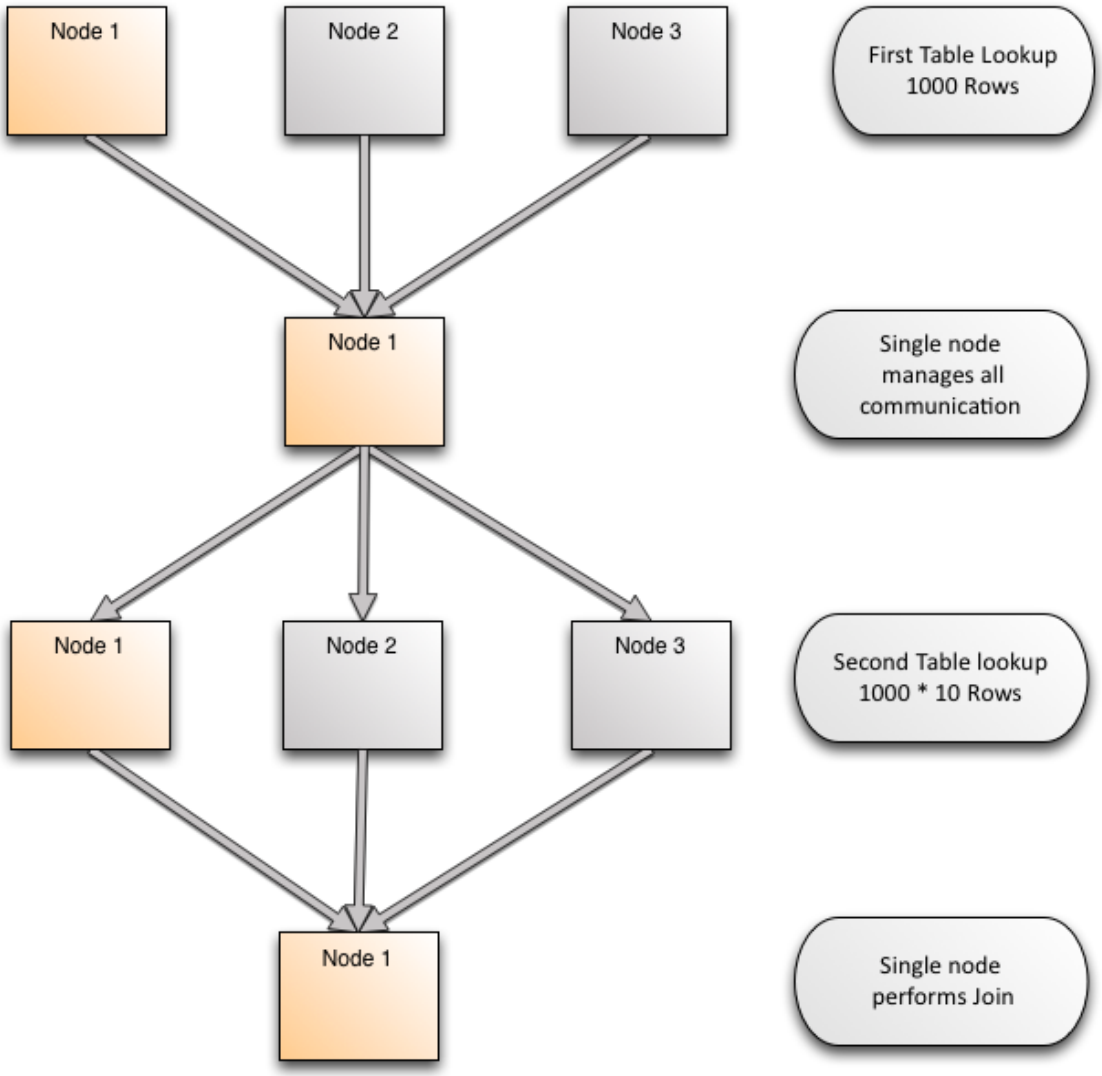
Joins with Broadcast (Competition)

None of our competitors have independent distribution for indexes. All sharding-like approaches distribute indexes according to the primary key. So, given an index on column like bundler_id in the previous example, the system has no clue where that key goes, since distribution was done based on donation.id. This leads to broadcasts, and therefore N way joins do not scale linearly. Notice in the image that broadcasts are shown by nodes getting rows (arrows) of their color - which are the ones that will find matching data for join, and of other colors - where the lookup will show no matching data. Oracle and MySql cluster do broadcasts like this.



Joins with Single Query Node (Competition)

ClustrixDB uses Massively Parallel Processing (MPP) to leverage multiple cores across nodes to make a single query go faster. Some competing products use a single node to evaluate the query, which leads to a limited speed-up of the query as the volume of data increases. This includes Oracle RAC and NuoDB, neither of which can use cores across multiple nodes to evaluate a join. All the data here needs to flow to a single node that evaluates a single query.



Most competitors are unable to match the linear scaling of joins that ClustrixDB provides. This is because they either broadcast in all cases (except when join is on primary key) or that only a single node ever works on evaluating a single query (other nodes send data over). This allows ClustrixDB to scale better than the competition, giving it a unique value proposition in transactional analytic space (OLAP).

Scaling Aggregates

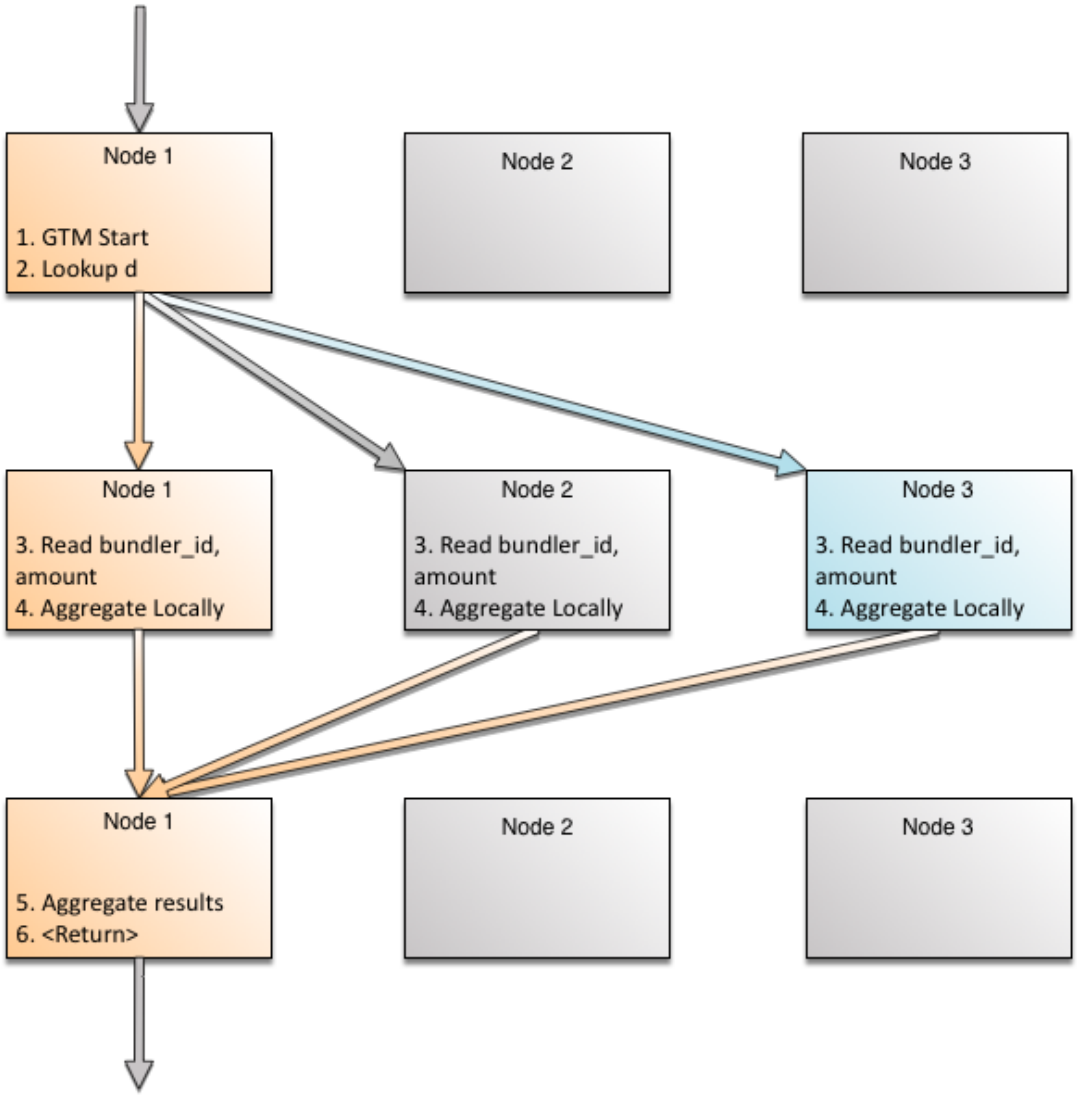
Scaling aggregates requires aggregating in parallel on multiple nodes. This leaves the final work of combining the partial aggregation from each node to a simple task on a much smaller data size.

This query is used to generate a report of the sum of donations by bundler_id. Here, it makes sense to join with the bundlers table as well as to get name, but we already understand how joins work - so let's keep the example simple.

```
sql> SELECT id, SUM(amount) FROM donation d GROUP BY by bundler_id;
```

Distributed Aggregates (ClustrixDB)

Here, the donation table is distributed on the donation.id field. If we were to group by this field, data from across nodes would not share same grouping key. This would then require no re-aggregation. But in this example, we show a case where the same value might be on different nodes (let's ignore that we can use the index). Below is how it will work in the most general case:



Here, we see that the data motion and sequential work is minimized for aggregation. We choose distributed aggregation when the data reduction is estimated to be good based on the statistics.

Aggregates with no Re-aggregation

Please note that we need to Aggregate results of local aggregates on GTM node, only if the values from different nodes may overlap. With complex queries involving joins, this is usually the case. Let's look at a simpler query:

```
sql> SELECT DISTINCT bundler_id FROM donation;
```

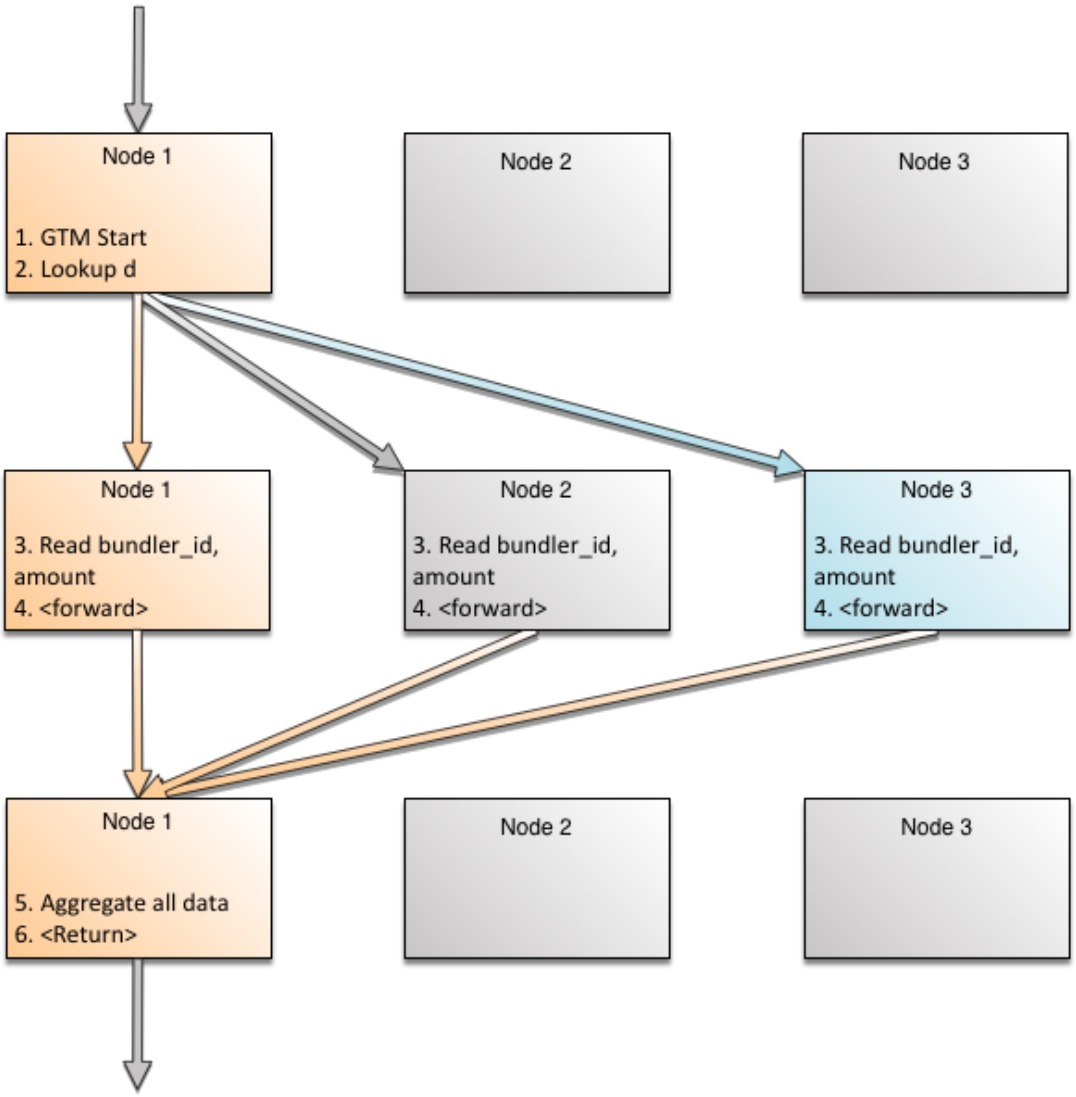
In this case, there is a secondary key (or index) called _bundler_key_donation, that is distributed by bundler_id. This has two implications:

- bundler_id values on each node are unique
- bundler_id values on each node are sorted

To efficiently implement this ClustrixDB will only need the distributed aggregation to hold one row at a time. So if a node reads bundler_id = 5, it will store that and forward it to GTM node. Then it will discard all subsequent values = 5 till it sees a new values 6. On the GTM node, there is no re-aggregation required since values from each node are unique, it merely merges the streams.

Single Node Aggregates (the Competition)

Most primary databases do not have Massively Parallel Processing, so all aggregation happens on the single node that is receiving the query. This is how Oracle, NuoDB and MongoDB work. This moves a lot more data, is unable to use the resources of more than one machine, and does not scale well.



Discussion

Let's go over the concepts that are involved in distributed evaluation and parallel processing of Queries.

Query Lifetime

User queries are distributed across nodes in ClustrixDB. Here is how a query is processed:

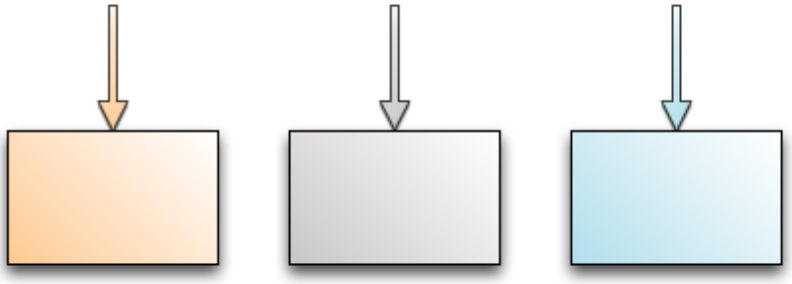
1. The query is associated with a session on the node where it arrives, and that node orchestrates the concurrency control for it.
2. After parsing, the query goes through the query optimizer and planner where an optimal plan is chosen for it based on the statistics.
3. The plan is then compiled by the compiler, which breaks it into smaller query fragments and runs code optimizations.
4. The compiled query fragments are then run in a distributed way across nodes, and the results are returned to the node where the query started and back to the user.

Using Pipeline and Concurrent Parallelism

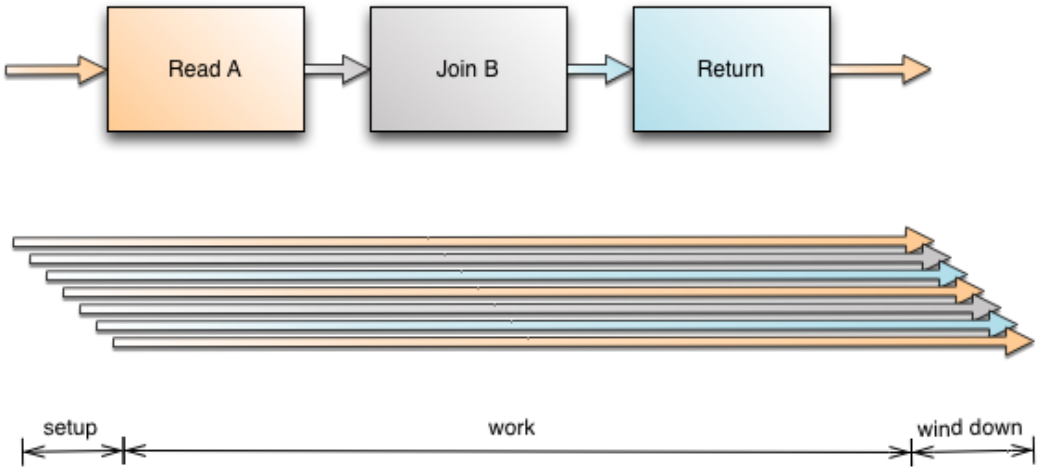
ClustrixDB uses both concurrent parallelism and pipeline parallelism for massively parallel computation of your queries. Simpler queries like point selects and inserts usually only use a single core on a node for its evaluation. As more nodes are added, more cores and memory are available, allowing ClustrixDB to handle more simple queries concurrently.

More complex analytic queries, such as those involving joins and aggregates, take advantage of both pipeline and concurrent parallelism, using cores from multiple nodes and multiple cores on a single node to evaluate queries fast. Take a look at the following examples how queries scale to better understand the parallelism.

Concurrent parallelism is between slices, each of which may be on one or more nodes.



Pipeline parallelism occurs among rows in a single query. Each row in an analytic query might involve three hops for a two-way join. The latency added to one row is the latency of three hops, so set-up time is negligible. The other possible limit is bandwidth. As previously mentioned, ClustrixDB has independent distribution for each table and index, so we only forward rows to the correct nodes.



How Do You Scale Evaluation in a Distributed System?

This table summarizes the different elements of a scalable distributed database as well as what it takes to build one:

	Throughput	**Speedup**	**Concurrent Writes**
Logical Requirement	Work increases linearly with • # of Queries	Analytic query speed increases linearly with • # of Nodes	Concurrency Overhead does not increase with • # of writes

System Characteristic	For evaluation of one query • # of messages (and work) don't increase with # nodes	For evaluation of one query • Multiple nodes should process it in parallel • Multiple cores within a node should process it in parallel • Every new node should add more memory to hold ever larger active working set	For evaluation of one query • Only one node owns each part of the data and writes to it • System is able to take fine-grained row level locks • Read and Write interference is remove through MVCC

Why Systems Fail	Many databases use sharding-like approach to slice their data
	• They co-locate indexes with base table
	• Then index lookups become broadcasts leading to the following effects:
	  • Nodes read broadcast messages to look up information
	  • They get data from disk
	  • They realize they don't have matching data
	  • This is wasted work
		Many databases do not use Massively Parallel Processing
		• Do not use more than one node to evaluate the query.
		• They might push down filters when fetching data from other nodes, but that is minimal.
		• Most work is co-ordinated or completed by a single node
			Many databases used Shared Data architecture where
			• Multiple nodes pull data from same storage node
			• Multiple nodes need to write to same data causing ping-pong of hot data
			• Database nodes take coarse-grained locks (DB-Level)

Examples of Limited Design	Colocation of indexes leading to broadcasts • MySql Server • MongoDB • Sql Sharding	Single Node Query Processing leading to limited query speedup • Oracle RAC • NuoDB • MongoDB	Shared data leading to ping-pong effect on hot data • Oracle RAC • NuoDB Coarse-grained locking leading to limited write concurrency • MongoDB
Why ClustrixDB Shines	ClustrixDB has • Independent distribution for tables and indexes • Queries involving any key use unicast messages • Results in near linear scaling of queries	ClustrixDB has • Massively parallel processing • Multiple cores within and across nodes work in parallel to speed up analytic queries	ClustrixDB has • Shared nothing architecture • Fine-grained row-level locking • MVCC to remove read-write interference

Conclusion

For simple and complex queries, ClustrixDB is able to scale almost linearly due to our query evaluation approach. We also have seen how most of our competitors hit various bottlenecks due to the design choices they have made.