Joins for Hybrid Warehouses: Exploiting Massive Parallelism in Hadoop and Enterprise Data Warehouses

Yuanyuan Tian[1], Tao Zou[2]*, Fatma Özcan[1], Romulo Goncalves[3]†, Hamid Pirahesh[1]
[1]*IBM Almaden Research Center, USA* {ytian, fozcan, pirahesh}@us.ibm.com
[2]*Google Inc, USA* taozou@google.com
[3]*Netherlands eScience Center, Netherlands* r.goncalves@esciencecenter.nl

ABSTRACT

HDFS has become an important data repository in the enterprise as the center for all business analytics, from SQL queries, machine learning to reporting. At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. This has created the need for a new generation of special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*. There are many applications that require correlating data stored in HDFS with EDW data, such as the analysis that associates click logs stored in HDFS with the sales data stored in the database. All existing solutions reach out to HDFS and read the data into the EDW to perform the joins, assuming that the Hadoop side does not have the efficient SQL support.

In this paper, we show that it is actually better to do most data processing on the HDFS side, provided that we can leverage a sophisticated execution engine for joins on the Hadoop side. We identify the best hybrid warehouse architecture by studying various algorithms to join database and HDFS tables. We utilize Bloom filters to minimize the data movement, and exploit the massive parallelism in both systems to the fullest extent possible. We describe a new *zigzag join* algorithm, and show that it is a robust join algorithm for hybrid warehouses which performs well in almost all cases.

1. INTRODUCTION

Through customer engagements, we observe that HDFS has become the *core storage system* for all enterprise data, including enterprise application data, social media data, log data, click stream data, and other Internet data. Enterprises are using various big data technologies to process this data and drive actionable insights. HDFS serves as the storage where other distributed processing frameworks, such as MapReduce [11] and Spark [43], access and operate on the large volumes of data.

At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. EDWs are usually shared-nothing parallel databases that support complex SQL processing, updates, and transactions. As a result, they manage up-to-date data and support various business analytics tools, such as reporting and dashboards.

Many new applications have emerged, requiring the access and correlation of data stored in HDFS and EDWs. For example, a company running an ad-campaign may want to evaluate the effectiveness of its campaign by correlating click stream data stored in HDFS with actual sales data stored in the database. These applications together with the co-existence of HDFS and EDWs have created the need for a new generation of special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*.

Many existing solutions [37, 38] to integrate HDFS and database data use utilities to replicate the database data onto HDFS. However, it is not always desirable to empty the warehouse and use HDFS instead, due to the many existing applications that are already tightly coupled to the warehouse. Moreover, HDFS still does not have a good solution to update data in place, whereas warehouses always have up-to-date data. Other alternative solutions either statically pre-load the HDFS data [41, 17, 18], or fetch the HDFS data at query time into EDWs to perform joins [14, 13, 28]. They all have the implicit assumption that SQL-on-Hadoop systems do not perform joins efficiently. Although this was true for the early SQL-on-Hadoop solutions, such as Hive [39], it is not clear whether the same still holds for the current generation solutions such as IBM Big SQL [19], Impala [20], and Presto [33]. There was a significant shift last year in the SQL-on-Hadoop solution space, where these new systems moved away from MapReduce to shared-nothing parallel database architectures. They run SQL queries using their own long-running daemons executing on every HDFS DataNode. Instead of materializing intermediate results, these systems pipeline them between computation stages. In fact, the benefit of applying parallel database techniques, such as pipelining and hash-based aggregation, have also been previously demonstrated by some alternative big data platforms to MapReduce, like Stratosphere [5], Asterix [6], and SCOPE [10]. Moreover, HDFS tables are usually much bigger than database tables, so it is not always feasible to ingest HDFS data and perform joins in the database. Another important observation is that enterprises are investing more on big data systems like Hadoop, and less on expensive EDW systems. As a result, there is more capacity on the Hadoop side. Remotely reading HDFS data into the database introduces significant overhead and burden on the EDWs because they are fully utilized by existing applications, and hence carefully monitored and managed.

Split query processing between the database and HDFS has been addressed by PolyBase [13] to utilize vast Hadoop resources. HDFS clusters usually run on cheaper commodity hardware and have much larger capacity than databases. However, PolyBase only considers

---

*Work described in this paper was done while the author was working at IBM

†Work described in this paper was done while the author was working at IBM

10.5441/002/edbt.2015.33

pushing down limited functionality, such as selections and projections, and considers pushing down joins only when both tables are stored in HDFS.

Federation [21, 2, 34, 40, 30] is a solution to integrate data stored in autonomous databases, while exploiting the query processing power of all systems involved. However, existing federation solutions use a client-server model to access the remote databases and move the data. In particular, they use JDBC/ODBC interfaces for pushing down a maximal sub-query and retrieving its results. Such a solution ingests the result data serially through the single JDBC/ODBC connection, and hence is only feasible for small amounts of data. In the hybrid warehouse case, a new solution that connects at a lower layer is needed to exploit the massive parallelism on both the HDFS side and the EDW side.

In this paper, we identify an architecture for hybrid warehouses by 1-) building a system that provides parallel data movement by exploiting the massive parallelism of both HDFS and EDWs to the fullest extent possible, and 2-) studying the problem of efficiently executing joins between HDFS and EDW data. We start by adapting well-known distributed join algorithms, and propose extensions that work well between a parallel database and an HDFS cluster. Note that these joins work across two heterogeneous systems and hence have asymmetric properties that needs to be taken into account.

HDFS is optimized by large bulk I/O, and as a result record level indexing does not provide significant performance benefits. Other means, like column-store techniques [1, 31, 29], need to be exploited to speed up data ingestion.

Parallel databases use various techniques to optimize joins and minimize data movement. They use broadcast joins when one of the tables participating in the join is small enough, and the other is very large, to save communication cost. The databases also exploit careful physical data organization for joins. They rely on query workloads to identify joins between large tables, and co-partition them on the join key to avoid data communication at query time.

In the hybrid warehouse, these techniques have limited applicability. Broadcast joins can only be used in limited cases, because the data involved is usually very large. As the database and the HDFS are two independent systems that are managed separately, co-partitioning related tables is also not an option. As a result, we need to adapt existing join techniques to optimize the joins between very large tables when neither is partitioned on the join key. It is also very important to note that no existing EDW solution in the market today has a good solution for joining two large tables when they are not co-partitioned.

We exploit Bloom filters to reduce the data communication costs in joins for hybrid warehouses. A Bloom filter is a compact data structure that allows testing whether a given value is in a set very efficiently, with controlled false positive rate. Bloom filters have been proposed in the distributed relational query setting [25]. But they are not used widely, because they introduce overhead of extra computation and communication. In this paper, we show that Bloom filters are almost always beneficial when communicating data in the hybrid warehouse which integrated two heterogeneous and massively parallel data platforms, as opposed to the homogeneous parallel databases. Furthermore, we describe a new join algorithm, the *zigzag join*, which uses Bloom filters both ways to ensure that only the records that will participate in the join need to be transferred through the network. The zigzag join is most effective when the tables involved in the join do not have good local predicates to reduce their sizes, but the join itself is selective.

In this work, we consider executing the join both in the database and on the HDFS side. We implemented the proposed join algorithms for the hybrid warehouse using a commercial shared-nothing parallel database with the help of user-defined functions (UDFs), and our own execution engine for joins on HDFS, called JEN. To implement JEN, we took a prototype of the I/O layer and the scheduler from an existing SQL-on-Hadoop system, IBM Big SQL [19], and extended it with our own runtime engine which is able to pipeline operations and overlay network communication with processing and data scanning. We observe that with such a sophisticated execution engine on HDFS, it is actually often better to execute the joins on the HDFS side.

The contributions of this paper are summarized as follows:

- Through detailed experiments, we show that it is often better to execute joins on the HDFS side as the data size grows, when there is a sophisticated execution engine on the HDFS side. To the best of our knowledge, this is the first work that argues for such a solution.

- We describe JEN, a sophisticated execution engine on the HDFS side to fully exploit the various optimization strategies employed by a shared-nothing parallel database architecture, including multi-threading, pipelining, hash-based aggregation, etc. JEN utilizes a prototype of the I/O layer and the scheduler from IBM Big SQL and provides parallel data movement between HDFS and an EDW to exploit the massive parallelism on both sides.

- We revisit the join algorithms that are used in distributed query processing, and adapt them to work in the hybrid warehouse between two heterogeneous massively parallel data platforms. We utilize Bloom filters, which minimize the data movement and exploit the massive parallelism in both systems.

- We describe a new join algorithm, the zigzag join, which uses Bloom filters on both sides, and provide a very efficient implementation that minimizes the overhead of Bloom filter computation and exchange. We show that the zigzag join algorithm is a robust algorithm that performs the best for hybrid warehouses in almost all cases.

The rest of the paper is organized as follows: We start with a concrete example scenario, including our assumptions, in Section 2. The join algorithms are discussed in Section 3. We implemented our algorithms using a commercial parallel database and our own join execution engine on HDFS. In Section 4, we describe this implementation. We provide detailed experimental results in Section 5, discuss related work in Section 6, and conclude in Section 7.

2. AN EXAMPLE SCENARIO

In this paper, we study the problem of joins in the hybrid warehouse. We will use the following example scenario to illustrate the kind of query workload we focus on. This example represents a wide range of real application needs.

Consider a retailer, such as Walmart or Target, which sells products in local stores as well as online. All the transactions, either offline or online, are managed and stored in a parallel database, whereas users' online click logs are captured and stored in HDFS. The retailer wants to analyze the correlation of customers' online behaviors with sales data. This requires joining the transaction table T in the parallel database with the log table L on HDFS. One such analysis can be expressed as the following SQL query.

SELECT L.url_prefix, **COUNT**(*)
FROM T, L
WHERE T.category = ‘Canon Camera’
AND region(L.ip)= ‘East Coast’
AND T.uid=L.uid
AND T.tdate $>=$ L.ldate **AND** T.tdate $<=$ L.ldate+1
GROUP BY L.url_prefix

This query tries to find out the number of views of the urls visited by customers with IP addresses from East Coast who bought Canon cameras within one day of their online visits.

Now, we look at the structure of the example query. It has local predicates on both tables, followed by an equi-join. The join is also coupled with predicates on the joined result, as well as group-by and aggregation. In this paper, we will describe our algorithms using this example query.

In common setups, a parallel database is deployed on a small number (10s to 100s) of high-end servers, whereas HDFS resides on a large number (100s to 10,000s) of commodity machines. We assume that the parallel database is a full-fledged shared-nothing parallel database. It has an optimizer, indexing support and sophisticated SQL engine. On the HDFS side, we assume a scan-based processing engine without any indexing support. This is true for all the existing SQL-on-Hadoop systems, such as MapReduce-based Hive [39], Spark-based Shark [42], and Impala [20]. We do not tie the join algorithm descriptions to a particular processing framework, thus we generalize any scan-based distributed data processor on HDFS as a HQP (HDFS Query Processor). For data characteristics, we assume that both tables are large, but the HDFS table is much larger, which is the case in most realistic scenarios. In addition, since we focus on analytic workloads, we assume there is always group-by and aggregation at the end of the query. As a result, the final query result is relatively small. Finally, without loss of generality, we assume that queries are issued at the parallel database side and the final results are also to be returned at the database side. Note that forwarding a query from the database to HDFS is relatively cheap, so is passing the final results from HDFS back to the database.

Note that in this paper we focus on the actual join algorithms for hybrid warehouses, thus we only include a two-way join in the example scenario. Real big data queries may involve joining multiple tables. For these cases, we need to rely on the query optimizer in the database to decide on the right join orders, since queries are issued at the database side in our setting. However, the study of the join orders in hybrid warehouse is beyond the scope of this paper.

3. JOIN ALGORITHMS

In this section, we describe a number of algorithms for joining a table stored in a shared-nothing parallel database with another table stored in HDFS. We start by adapting well-known distributed join algorithms, and explore ways to minimize the data movement between these two systems by utilizing Bloom filters. While existing approaches [27, 26, 25, 24, 32] were designed for homogeneous environments, our join algorithms work across two heterogeneous systems in the hybrid warehouse. When we design the algorithms, we strive to leverage the processing power of both systems and maximize parallel execution.

Before we describe the join algorithms, let’s provide a brief introduction to Bloom filters first. A Bloom filter is essentially a bit array of m bits with k hash functions defined to summarize a set of elements. Adding an element to the Bloom filter involves applying the k hash functions on the element and setting the corresponding positions of the bit array to 1. Symmetrically, testing whether an element belongs to the set requires simply applying the hash functions and checking whether all of the corresponding bit positions are set to 1. Obviously, the testing incurs some false positives. However, the false positive rate can be computed based on m, k and n, where n is the number of unique elements in the set. Therefore, m and k can be tuned for desired false positive rate. Bloom filter is a compact and efficient data structure for us to take advantage of the join selectivity. By building a Bloom filter on the join keys of one table, we can use it to prune out the non-joinable records from the other table.

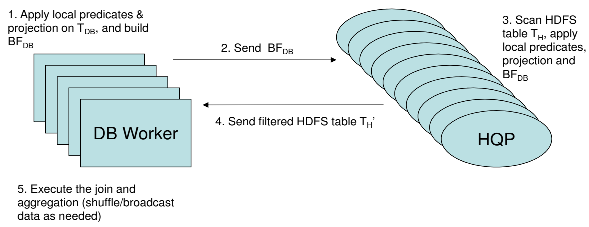


Figure 1: Data flow of DB-side join with Bloom filter

3.1 DB-Side Join

Many database/HDFS hybrid systems, including Microsoft Polybase [13], Pivotal HAWQ [15], TeraData SQL-H [14], and Oracle Big Data SQL [28], fetch the HDFS table and execute the join in the database. We first explore this approach, which we call DB-side join. In the plain version, the HDFS side applies local predicates and projection, and sends the filtered HDFS table in parallel to the database. The performance of this join method is dependent on the amount of data that needs to be transferred from HDFS. Two factors determine this size: the selectivity of the local predicates over the HDFS table and the size of the projected columns.

Note that the HDFS table is usually much larger than the database table. Even if the local predicates are highly selective, the filtered HDFS table can still be quite large. In order to further reduce the amount of data transferred from HDFS to the parallel database, we introduce a Bloom filter on the join key of the database table after applying local predicates, and send the Bloom filter to the HDFS side. This technique enables the use of the join selectivity to filter out HDFS records that cannot be joined. This DB-side join algorithm is illustrated in Figure 1.

In this DB-side join algorithm, each parallel database node (DB worker in Figure 1) first computes the Bloom filter for their local partitions and then aggregate them into a global Bloom filter (BF_{DB}) by simply applying bitwise OR. We take advantage of the query optimizer of the parallel database. After the filtered HDFS data is brought into the database, it is joined with the database data using the join algorithm (broadcast or repartition) chosen by the query optimizer. Note that in the DB-side join, the HDFS data may need to be shuffled again at the database side before the join (e.g. if repartition join is chosen by the optimizer), because we do not have access to the partitioning hash function of the database.

In the above algorithm, there are different ways to send the database Bloom filter to HDFS and transmit the HDFS data to the database. Which approach works best depends on the network topology and the bandwidth. We defer the discussion of detailed implementation choices to Section 4.

3.2 HDFS-Side Broadcast Join

The second algorithm is called HDFS-side broadcast join, or simply broadcast join. This is the first algorithm that executes the

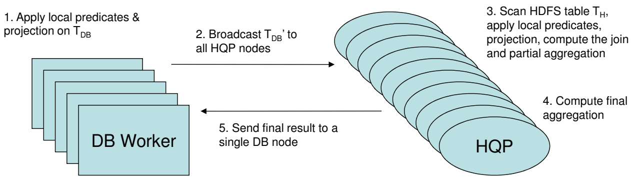


Figure 2: Data flow of HDFS-side broadcast join

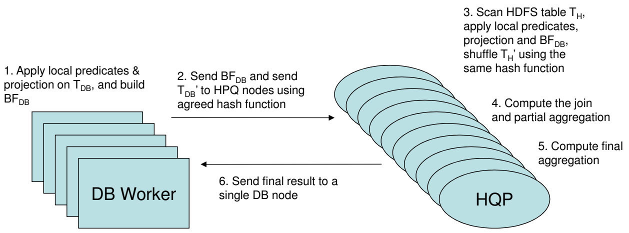


Figure 3: Data flow of HDFS-side repartition join with Bloom filter

join on the HDFS side. The rational behind this algorithm is that if the predicates on the database table are highly selective, the filtered database data is small enough to be sent to every HQP node, so that only local joins are needed without any shuffling of the HDFS data. When the join is executed on the HDFS side, it is logical to push-down the grouping and aggregation to the HDFS side as well. This way, only a small amount of summary data needs to be transferred back to the database to be returned to the user. The HDFS-side broadcast join algorithm is illustrated in Figure 2.

In the first step, each database node applies local predicates and projection over the database table. Each database node broadcasts its filtered partition to every HQP node (Step 2). Each HQP node performs a local join in Step 3. Group-by and partial aggregation are also carried out on the local data in this step. The final aggregation is computed in Step 4 and sent to the database in Step 5.

3.3 HDFS-Side Repartition Join

The second HDFS-side algorithm we consider is the HDFS-side repartition join, or simply repartition join. If the local predicates over the database table are not highly selective, then broadcasting the filtered data to all HQP nodes will not be a good option. In this case, we need a robust join algorithm. We expect the HDFS table to be much larger than the database table in practice, and hence it makes more sense to transfer the smaller database table and execute the final join at the HDFS side. Just as in the DB-side join, we can also improve this basic version of repartition join by introducing a Bloom filter. Figure 3 demonstrates this improved algorithm.

In Step 1, all database nodes apply local predicates over the database table, and project out the required columns. All database nodes also compute their local Bloom filters which are then aggregated into a global Bloom filter and sent to the HQP nodes. In this algorithm, the HDFS side and the database agree on the hash function to use when shuffling the data. In Step 2, all database nodes use this agreed hash function and send their data to the identified HQP nodes. This means that once the database data reaches the HDFS side, it doesn't need to be re-shuffled among the HQP nodes. In Step 3 of the HDFS-side repartition join, all HQP nodes apply the local predicates and projection over the HDFS table as well as the Bloom filter sent by the database. The Bloom filter further filters out the HDFS data. The HQP nodes use the same hash function to shuffle the filtered HDFS table. Then, they perform the join and partial aggregation (step 4). The final aggregation is executed on the HDFS side in Step 5 and sent to the database in Step 6.

3.4 HDFS-Side Zigzag Join

When local predicates on neither the HDFS table nor the database table are selective, we need to fully exploit the join selectivity to perform the join efficiently. In some sense, a selective join can be used as if it were extended local predicates on both tables. To illustrate this point, let's first introduce the concepts of *join-key selectivity* and *join-key predicate*.

Let T'_{DB} be the table after local predicates and projection on the database table T_{DB}, and T'_H be the table after local predicates and projection on the HDFS table T_H. We define $JK(T'_{DB})$ as the set of join keys in T'_{DB}, and $JK(T'_H)$ as the set of join keys in T'_H. We know that only the join keys in $JK(T'_{DB}) \cap JK(T'_H)$ will appear in the final join result. So, only $\frac{JK(T'_{DB}) \cap JK(T'_H)}{JK(T'_H)}$ fraction of the unique join keys in T'_H will participate in the join. We call this fraction the *join-key selectivity* on T'_H, denoted as $S_{T'_H}$. Likewise, the join-key selectivity on T'_{DB} is $S_{T'_{DB}} = \frac{JK(T'_{DB}) \cap JK(T'_H)}{JK(T'_{DB})}$. Leveraging the join-key selectivities through Bloom filters is essentially like applying extended local predicates on the join key columns of both tables. We call them *join-key predicates*.

Through the use of a 1-way Bloom filter, the DB-side join and the repartition join described in previous sections are only able to leverage the HDFS-side join-key predicate to reduce either the HDFS data transferred to the database or the HDFS data shuffled among the HQP workers. The DB-side join-key predicate is not utilized at all. Below, we introduce a new algorithm, zigzag join, to fully utilize the join-key predicates on both sides in reducing data movement, through the use of 2-way Bloom filters. Again, we expect the HDFS table to be much larger than the database table in practice, hence the final join in this algorithm is executed on the HDFS side, and both sides agree on the hash function to send data to the correct HQP nodes for the final join.

The zigzag join algorithm is described in Figure 4. In Step 1, all database nodes apply local predicates and projection, and compute their local Bloom filters. The database then computes the global Bloom filter BF_{DB} and sends it to all HQP nodes in Step 2. Like in the repartition join with Bloom filter, this Bloom filter helps reduce the amount of HDFS data that needs to be shuffled.

In Step 3, all HQP nodes apply their local predicates, projection and the database Bloom filter BF_{DB} over the HDFS table, and compute a local Bloom filter for the HDFS table. The local Bloom filters are aggregated into a global one, BF_H, which is sent to all database nodes. At the same time, the HQP nodes shuffle the filtered HDFS table based on the agreed hash function. In Step 5, the database nodes receive the HDFS Bloom filter BF_H and apply it to the database table to further reduce the number of database records that need to be sent. The application of Bloom filters on both sides ensure that only the data that will participate in the join (subject to false positive of the Bloom filter) needs to be transferred.

Note that in Step 5 the database data need to be accessed again. We rely on the advanced database optimizer to choose the best strategy: either to materialize the intermediate table $T_{DB'}$ after local predicates and projection are applied, or to utilize indexes to access the original table T_{DB}. It is also important to note that while the

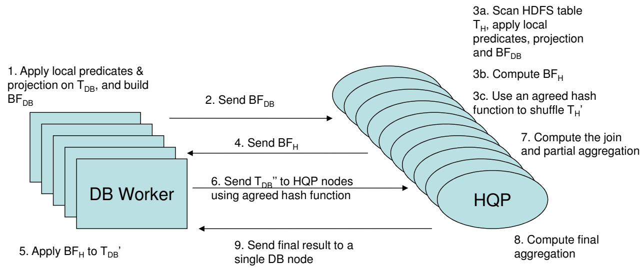


Figure 4: Data flow of zigzag join

HDFS bloom filter is applied to the database data, the HQP nodes are shuffling the HDFS data in parallel, hence overlapping many steps of the execution.

In Step 6, the database nodes send the further filtered database data to the HQP nodes using the agreed hash function. The HQP nodes perform the join and partial aggregation (Step 7), collaboratively compute the global aggregation (Step 8), and finally send the result to the database (Step 9).

Note that zigzag join is the only join algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides. The HDFS data shuffled across HQP nodes are filtered by the local predicates on T_H, the local predicates on T_{DB} (as BF_{DB} is built on T_{DB} after local predicates), and the join-key predicate on T'_H. Similarly, the database records transferred to the HDFS side are filtered by the local predicates on T_{DB}, the local predicates on T_H (as BF_H is built on T_H after local predicates), and the join-key predicate on T'_{DB}.

Although Bloom filters and semi-join techniques are known in the literature, they are not widely used in practice due to the overhead of computing Bloom filters and multiple data scans. However, the asymmetry of slow HDFS table scan and fast database table access makes these techniques more desirable in a hybrid warehouse. Note that a variant version of the zigzag join algorithm which executes the final join on the database side will not perform well, because scanning the HDFS table twice, without the help of indexes, is expected to introduce significant overhead.

4. IMPLEMENTATION

In this section, we provide an overview of our implementation of the join algorithms for the hybrid warehouse and highlight some important details.

4.1 Overview

In our implementation, we used IBM DB2 Database Partitioning Feature (DPF), which is a shared-nothing distributed version of DB2, as our EDW. We implemented all the above join algorithms using C user-defined functions (UDFs) in DB2 DPF and our own C++ MPI-based join execution engine on HDFS, called JEN. JEN is our specialized implementation of HQP used in the algorithm descriptions in Section 3. We used a propotype of the I/O layer and the scheduler from an early version of IBM Big SQL 3.0 [19], and build JEN on top of them. We also utilized Apache HCatalog [16] to store the meta data of the HDFS tables.

JEN consists of a single coordinator and a number of workers, with each worker running on an HDFS DataNode. JEN workers are responsible for reading parts of HDFS files, executing local query plans, and communicating with other workers, the coordinator, and DB2 DPF workers. Each JEN worker is multi-threaded, capable of exploiting all the cores on a machine. The communication between two JEN workers or with the coordinator is done through TCP/IP sockets. The JEN coordinator has multiple roles. First, it is responsible for managing the JEN workers and their state so that workers know which other workers are up and running in the system. Second, it serves as the central contact for the JEN workers to learn the IPs of the DB2 workers and vice versa, so that they can establish communication channels for data transfers. Third, it is also responsible for retrieving the meta data (HDFS path, input format, etc) for HDFS tables from HCatalog. Once the coordinator knows the path of the HDFS table, it contacts the HDFS NameNode to get the locations of each HDFS block, and evenly assigns the HDFS blocks to the JEN workers to read, respecting data locality.

At the DB2 side, we utilized the existing database query engine as much as possible. For the functionalities not provided, such as computing and applying Bloom filters, and different ways of transferring data to and from JEN workers, we implemented them using *unfenced* C UDFs, which provide performance close to built-in functions as they run in the same process as the DB2 engine. The communication between a DB2 DPF worker and a JEN worker is also through TCP/IP sockets. Note that to exploit the multi-cores on a machine we set up multiple DB2 workers on each machine of a DB2 DPF cluster, instead of one DB2 worker enabled with multi-core parallelism. This is mainly to simplify our C UDF implementations, as otherwise we have to deal with intra-process communications inside a UDF.

Each of the join algorithms is invoked by issuing a *single* query to DB2. With the help of UDFs, this single query executes the *entire* join algorithm: initiating the communication between the database and the HDFS side, instructing the two sides to work collaboratively, and finally returning the results back to the user.

4.1.1 The DB-Side Join Example

Let's use an example to illustrate how the database side and the HDFS side collaboratively execute a join algorithm. If we want to execute the example query in Section 2 using the DB-side join with Bloom filter, we submit the following SQL query to DB2.

```
with LocalFilter(lf) as (
select get_filter(max(cal_filter(uid))) from T
where T.category='Canon Camera'
group by dbpartitionnum(tid)
),
GlobalFilter(gf) as (
select * from
table(select combine_filter(lf) from LocalFilter)
where gf is not null
),
Clicks(uid, url_prefix, ldate) as (
select uid, url_prefix, ldate
from GlobalFilter,
table(read_hdfs('L', 'region(ip)= \'East Coast\'',
'uid, url_prefix, ldate', GlobalFilter.gf, 'uid'))
)
select url_prefix, count(*) from Clicks, T
where T.category='Canon Camera' and Clicks.uid=T.uid
and days(T.tdate)-days(Clicks.ldate)>=0
and days(T.tdate)-days(Clicks.ldate)<=1
group by url_prefix
```

In the above SQL query, we assume that the database table T is distributed across multiple DB2 workers on the `tid` field. The first sub query (`LocalFilter`) uses two scalar UDFs `cal_filter` and `get_filter` together to compute a Bloom filter on the local partition of each DB2 worker We enabled the two UDFs to execute in parallel, and the statement `group by dbpartitionnum(tid)` further makes sure that each DB2 worker computes the Bloom filter

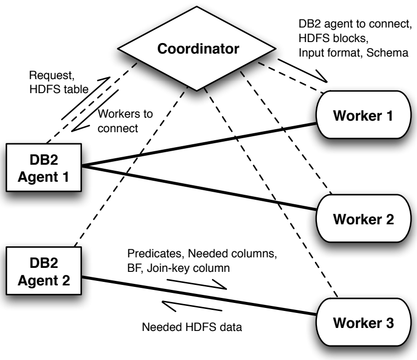


Figure 5: Communication in the `read_hdfs` UDF of the DB-side join with Bloom filter

on its local data in parallel. The second sub query (`GlobalFilter`) uses another scalar UDF `combine_filter` to combine the local Bloom filters into a single global Bloom filter (there is only one record which is the global Bloom filter returned for `GlobalFilter`). By declaring `combine_filter` "disallow parallel", we make sure it is executed once on one of the DB2 workers (all local Bloom filters are sent to a single DB2 worker). In the third sub query (`Clicks`), a table UDF `read_hdfs` is used to pass the following information to the HDFS side: the name of the HDFS table, the local predicates on the HDFS table, the projected columns needed to be returned, the global database Bloom filter, and the join-key column that the Bloom filter needs to be applied. In the same UDF, the JEN workers subsequently read the HDFS table and send the required data after applying predicates, projection and the Bloom filter back to the DB2 workers. The `read_hdfs` UDF is executed on each DB2 worker in parallel (the global Bloom filter is broadcast to all DB2 workers) and carries out the parallel data transfer from HDFS to DB2. After that, the join together with the group-by and aggregation is executed at the DB2 side. We pass a hint of the cardinality information to the `read_hdfs` UDF, so that the DB2 optimizer can choose the right plan for the join. The final result is returned to the user at the database side.

Now let's look into the details of the `read_hdfs` UDF. Since there is only one record in `GlobalFilter`, this UDF is called once per DB2 worker. When it is called on each DB2 worker, it first contacts the JEN coordinator to request for the connection information to the JEN workers. In return, the coordinator tells each DB2 worker which JEN worker(s) to connect to, and notifies the corresponding JEN workers to prepare for the connections from the DB2 workers. This process is shown in Figure 5. Without the loss of generality, let's assume that there are m DB2 workers and n JEN workers, and that $m \leq n$. For the DB-side join, the JEN coordinator evenly divides the n workers into m groups. Each DB2 worker establishes connections to all the workers in one group, as illustrated in Figure 5. After all the connections are established, each DB2 worker multi-casts the predicates on the HDFS table, the required columns from the HDFS table, the database Bloom filter and the join-key column to the corresponding group of JEN workers. At the same time, DB2 workers tell the JEN coordinator which HDFS table to read. The coordinator contacts the HCatalog to retrieve the paths of the corresponding HDFS files and the input format, and inquires the HDFS NameNode for the storage locations of the HDFS blocks. Then, the coordinator assigns the HDFS blocks and sends the assignment as well as the input format to the workers. After receiving all the necessary information, each JEN worker is

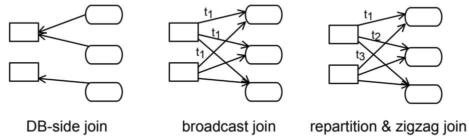


Figure 6: Data transfer patterns between DB2 workers and JEN workers in the join algorithms

ready to scan its share of the HDFS data. As it scans the data, it directly applies the local predicates and the Bloom filter from the database side, and sends the records with required columns back to its corresponding DB2 worker.

4.2 Locality-Aware Data Ingestion from HDFS

As our join execution engine on HDFS is scan-based, efficient data ingestion from HDFS is crucial for performance. We purposely deploy the JEN workers on all HDFS DataNodes so that we can leverage data locality when reading. In fact, when the JEN coordinator assigns the HDFS blocks to workers, it carefully considers the locations of each HDFS block to create balanced assignments and maximize the locality of data in a best-effort manner. Using this locality-aware data assignment, each JEN worker mostly reads data from local disks. We also enabled short-circuit reads for HDFS DataNodes to improve the local read speed. In addition, our data ingestion component uses multiple threads when multiple disks are used for each DataNode to further boost the data ingestion throughput.

4.3 Data Transfer Patterns

In this subsection, we discuss the data transfer patterns of different join algorithms. There are three types of data transfers that happen in all the join algorithms: among DB2 workers, among JEN workers, and between DB2 workers and JEN workers. For the data transfers among DB2 workers, we simply rely on DB2 to choose and execute the right transfer mechanisms. Among the JEN workers, there are three places that data transfers are needed: (1) shuffle the HDFS data for the repartition-based join in the repartition join (with/without Bloom filter) and the zigzag join, (2) aggregate the global HDFS Bloom filter for the zigzag join, and (3) compute the final aggregation result from the partial results on JEN workers in the broadcast join, the repartition join and the zigzag join. For (1), each worker simply maintains TCP/IP connections to all other workers and shuffles data through these connections. For (2) and (3), each worker sends the local results (either local Bloom filter or local aggregates) to a single designated worker chosen by the coordinator to finish the final aggregation.

The more interesting data transfers happen between DB2 workers and JEN workers. Again, there are three places that the data transfer is needed: shipping the actual data (HDFS or database), sending the Bloom filters, and transmitting the final aggregated results to the database for all the HDFS-side joins. Bloom filters and final aggregated results are much smaller than the actual data, how to transfer them has little impact on the overall performance. For the database Bloom filter sent to HDFS, we multi-cast the database Bloom filters to HDFS following the mechanism shown in Figure 5. For the HDFS Bloom filter sent to the database, we broadcast the HDFS Bloom filter from the designated JEN worker to all the DB2 workers. The final results on HDFS is simply transmitted from the designated JEN worker to a designated DB2 worker. In contrast to the above, we put more thoughts on how to ship the actual data between DB2 and HDFS. Figure 6 demonstrates the different patterns for transferring the actual data in the different join algorithms.

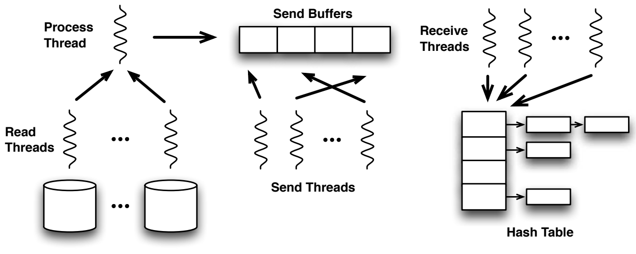


Figure 7: Interleaving of scanning, processing and shuffling of HDFS data in zigzag join

DB-side join with/without Bloom filter. For the two DB-side joins, we randomly partition the set of JEN workers into m roughly even groups, where m is the number of DB2 workers, then let each DB2 worker bring in the part of HDFS data in parallel from the corresponding group of JEN workers. DB2 can choose whatever algorithms for the final join that it sees fit based on data statistics. For example, when the database data is much smaller than HDFS data, the optimizer chooses to broadcast the database table for the join. When the HDFS data is much smaller than the database data, broadcasting the HDFS data is used. In the other cases, a repartition-based join algorithm is chosen. This means that when the HDFS data is transferred to the database side, it may need to be shuffled again among the DB2 workers. To avoid this second data transfer, we would have to expose the partitioning scheme of DB2 to JEN and teach the DB2 optimizer that the data received from JEN workers has already been partitioned in the desired way. Our implementation does not modify the DB2 engine, so we stick with this simpler and non-invasive data transfer scheme for the DB-side joins.

Broadcast join. There are multiple ways to broadcast the database data to JEN workers. One way is to let each DB2 worker connect to all the JEN workers and deliver its data to every worker. Another way is to have each DB2 worker only transfer its data to one JEN worker, which further passes on the data to all other workers. The second approach puts less stress on the inter-connection between DB2 and HDFS, but introduces a second round of data transfer among the JEN workers. We found empirically that broadcast join only works better than other algorithms when the database table after local predicates and projection is very small. For that case, even the first transfer pattern does not put much strain on the inter-connection between DB2 and HDFS. Furthermore, the second approach actually introduces extra latency because of the extra round of data transfer. For the above reasons, we use the first data transfer scheme in our implementation of the broadcast join.

Repartition join with/without Bloom filter and zigzag join. For these three join algorithms, the final join happens at the HDFS side. We expose the hash function for the final repartition-based join in JEN (DB2 workers can get this information from the JEN coordinator). When a database record is sent to the HDFS side, the DB2 worker uses the hash function to identify the JEN worker to send to directly.

4.4 Pipelining and Multi-Threading in JEN

In the implementation of JEN, we try to pipeline operations and parallelize computation as much as possible. Let's take the sophisticated zigzag join as an example.

At the beginning, every JEN worker waits to receive the global Bloom filter from DB2, which is a blocking operation, since all the remaining operations depend on this Bloom filter. After the Bloom filter is obtained, each worker starts to read its portion of the HDFS table (mostly from local disks) immediately. The data ingestion component is able to dedicate one *read thread* per disk when multiple disks are used for an HDFS DataNode. In addition, a separate *process thread* is used to parse the raw data into records based on the input format and schema of the HDFS table. Then it applies the local predicates, projection and the database Bloom filter on each record. For each projected record that passes all the conditions, this thread uses it to populate the HDFS-side Bloom filter, and applies the shuffling hash function on the join key to figure out which JEN worker this record needs to be sent to for the repartition-based join. Then, the record is put in a send buffer ready to be sent. All the above operations on a record are pipelined inside the process thread. At the same time, a pool of *send threads* poll the sending buffers to carry out the data transfers. Another pool of *receive threads* simultaneously receive records from other workers. And for each record received in a receive thread, it uses the record to build the hash table for the join. The multi-threading in this stage of the zigzag join is illustrated in Figure 7. As can be seen, scanning, processing and shuffling (sending and receiving) of HDFS data are carried out totally in parallel. In fact, the repartition join (with/without Bloom filter) also shares the similar interleaving of scanning, processing and shuffling of HDFS data. Note that reading from HDFS and shuffling data through networks are expensive operations, although we only have one process thread which applies the local predicates, Bloom filter and the projection, it is never the bottleneck.

As soon as the reading from HDFS finishes (read threads are done), a local Bloom filter is built on each worker. The workers send local Bloom filters to a designated worker to compute the global Bloom filter and pass it on to the DB2 workers. After that, every worker waits to receive and buffer the data from DB2 in the background. Once the local hash table is built (the send and receive threads in Figure 7 are all done), the received database records are used to probe the hash table, produce join results, and subsequently apply a hash-based group-by and aggregation immediately. Here again, all the operations on a database record are pipelined. When all the local aggregates are computed, each worker sends its partial result to a designated worker, which computes the final aggregate and sends to a single DB2 worker to return to the user.

Note that in our implementation of the zigzag join, we choose to build the hash table from the filtered HDFS data and use the transferred database data to prob the hash table for the final join, although the database data are expected to be smaller in most cases. This is because the filtered HDFS data is already being received during the scan of the HDFS table due to multi-threading. Empirically, we find that the receiving of the HDFS data is usually done soon after the scan is finished. On the other hand, the database data will not start to arrive until the HDFS table scan is done, as the HDFS side bloom filter is fully constructed only after all HDFS data are processed. Therefore, it makes more sense to start building the hash table on the filtered HDFS data while waiting for the later arrival of the database data. The current version of JEN requires that all data fit in memory for the local hash-based join on each worker. In the future, we plan to support spilling to disk to over come this limitation.

5. EXPERIMENTAL EVALUATION

Experimental Setup. For the HDFS cluster, we used 31 IBM System x iDataPlex dx340 servers. Each consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors (8 cores in total), 32GB RAM, 5x DATA disks and interconnected using 1Gbit Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24) and

Java 1.6. One server was dedicated as the NameNode, whereas the other 30 were used as DataNodes. We reserved 1 disk for the OS, and the remaining 4 for HDFS on each DataNode. HDFS replication factor is set to 2. A JEN worker was run on each DataNode and the JEN coordinator was run on the Namenode. For DB2 DPF, we used 5 servers. Each had 2x Intel Xeon CPUs @ 2.20GHz, with 6x physical cores each (12 physical cores in total), 12x SATA disks, 1x 10 Gbit Ethernet card, and a total of 96GB RAM. Each node runs 64-bit Ubuntu Linux 12.04, with a Linux Kernel version 3.2.0-23. We ran 6 database workers on each server, resulting in a total of 30 DB2 workers. 11 out of the 12 disks on each server were used for DB2 data storage. Finally, the two clusters were connected by a 20 Gbit switch.

Dataset. We generated synthetic datasets in the context of the example query scenario described in Section 2. In particular, we generated a transaction table T of 97GB with 1.6 billion records stored in DB2 DPF and a log table L on HDFS with about 15 billion records. The log table is about 1TB when stored in text format. We also stored the log table in the Parquet columnar format [31] with Snappy compression [36], to more efficiently ingest data from HDFS. The I/O layer of our JEN workers is able to push down projections when reading from this columnar format. The 1TB text log data is reduced to about 421GB in Parquet format. By default, our experiments were run on the Parquet formatted data, but in Section 5.4, we will compare Parquet format against text format to study their effect on performance. The schemas of the transaction table and the log table are listed below.

```
T(uniqKey bigint, joinKey int, corPred int, indPred
int, predAfterJoin date, dummy1 varchar(50), dummy2
int, dummy3 time)

L(joinKey int, corPred int, indPred int, predAfterJoin
date, groupByExtractCol varchar(46), dummy char(8))
```

The transaction table T is distributed on a unique key, called uniqKey, across the DB2 workers. The two tables are joined on a 4-byte int field joinKey. In both tables, there is one int column correlated with the join key called corPred, and another int column independent of the join key called indPred. They are used for local predicates. The date fields, named predAfterJoin, on the two tables are used for the predicate after the join. The varchar column groupByExtractCol in L is used for group-by. The remaining columns in each table are just dummy columns. Values of all fields in the two tables are uniformly distributed. The query that we ran in our experiments can be expressed in SQL as follows.

```
select extract_group(L.groupByExtractCol), count(*)
from T, L
where T.corPred<=a and T.indPred<=b
and L.corPred<=c and L.indPred<=d
and T.joinKey=L.joinKey
and days(T.predAfterJoin)-days(L.predAfterJoin)>=0
and days(T.predAfterJoin)-days(L.predAfterJoin)<=1
group by extract_group(L.groupByExtractCol)
```

In the above query, the local predicates on T and L are on the combination of the corPred and the indPred columns, so that we can change the join selectivities given the same selectivities of the combined local predicates. In particular, by modifying constants a and c, we can change the number of join keys participating in the final join from each table; but we can also modify the constants b and d accordingly so that the selectivity of the combined predicates stay intact for each table. We apply a UDF (extract_group) on the varchar column groupByExtractCol to extract an int column as the group-by column for the final aggregate count(*). To fully exploit

	HDFS tuples shuffled	DB tuples sent
repartition	5,854 million	165 million
repartition(BF)	591 million	165 million
zigzag	591 million	30 million

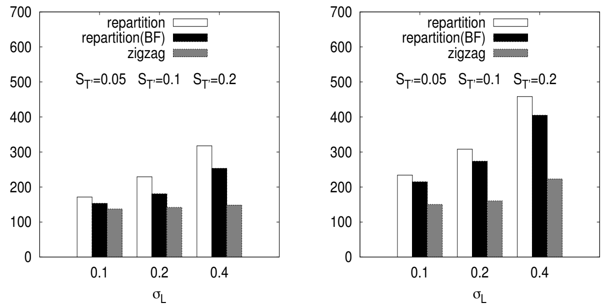
Table 1: Zigzag join vs repartition joins ($\sigma_T = 0.1$, $\sigma_L = 0.4$, $S_{L'} = 0.1$, $S_{T'} = 0.2$): # tuples shuffled and sent

the SQL support in DB2, we build one index on (corPred, indPred) and another index on (corPred, indPred, joinKey) of table T. The second index enables calculations of Bloom filters on T using an index-only access plan.

There are 16 million unique join keys in our dataset, so we create Bloom filters of 128 million bits (16MB) using 2 hash functions, which provides roughly 5% false positive rate. Note that exploring the different combinations of Bloom filter size and number of hash functions have been well studied before [9] and is beyond the scope of this paper. Our particular choice of the parameter values gave us good performance results in our experiments.

Our experiments were the only workloads that ran on the DPF cluster and the HDFS cluster. But, we purposely allocated less resources to the DPF cluster to mimic the case that the database is more heavily utilized. For all the experiments, we reported the warm-run performance numbers (we ran each experiments multiple times and excluded the first run when taking average).

In all the figures shown below, we denote the database table T after local predicates and projection as T' (predicate selectivity denoted as σ_T), and the HDFS table L after local predicates and projection as L' (predicate selectivity denoted as σ_L). We further represent the join-key selectivity on T' as $S_{T'}$ and the join-key selectivity on L' as $S_{L'}$.



(a) $\sigma_T = 0.1, S_{L'} = 0.1$ (b) $\sigma_T = 0.2, S_{L'} = 0.2$

Figure 8: Zigzag join vs repartition joins: execution time (sec)

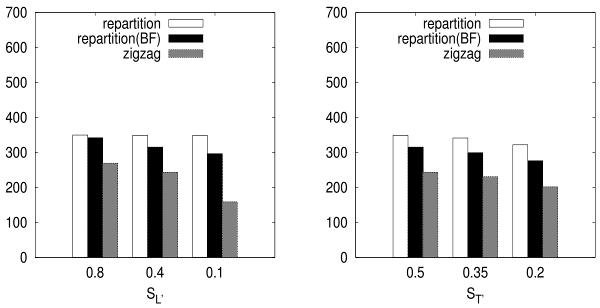
5.1 HDFS-Side Joins

We first study the HDFS-side join algorithms. We start with demonstrating the superiority of our zigzag join to the other reparation-based joins and then investigate when to use the broadcast join versus the repartition-based joins.

5.1.1 Zigzag Join vs Repartition Joins

We now compare the zigzag join to the repartition joins with and without Bloom filter. All three repartition-based join algorithms are best used when local predicate selectivities on both database and HDFS tables are low.

Figure 8 compares the execution times of the three algorithms with varying predicate and join-key selectivities on the Parquet formatted log table. It is evident that the zigzag join is the most efficient among the all repartition-based joins. It is up to 2.1x faster than the repartition join without Bloom filter and up to 1.8x faster than the repartition join with Bloom filter. When we zoom in the last three bars in Figure 8(a), Table 1 details the number of HDFS tuples shuffled across the JEN workers as well as the number of database tuples sent to the HDFS side for the three algorithms. The zigzag join is able to cut down the shuffled HDFS data by roughly 10x (corresponding to $S_{L'} = 0.1$) and the transferred database data by around 5x (corresponding to $S_{T'} = 0.2$). It is the only algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides. In Figure 9, we fix the predicate selectivities $\sigma_T = 0.1$ and $\sigma_L = 0.4$ to explore the effect of different join-key selectivities $S_{L'}$ and $S_{T'}$ on the three algorithms. As expected, with the same size of T' and L', the performance of zigzag join improves with when the join-key selectivity $S_{L'}$ or $S_{T'}$ decreases.



(a) $S_{T'} = 0.5$ (b) $S_{L'} = 0.4$

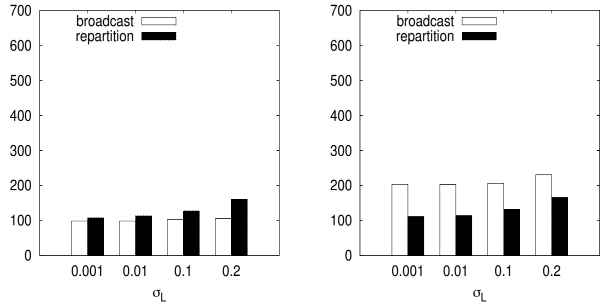
Figure 9: Zigzag join ($\sigma_T = 0.1$, $\sigma_L = 0.4$) with different $S_{L'}$ and $S_{T'}$ values: execution time (sec)

5.1.2 *Broadcast Join vs Repartition Join*

Besides the three repartition-based joins studied above, broadcast join is another HDFS-side join. To find out when this algorithm works best, we compare broadcast join and the repartition join without Bloom filter in Figure 10. We do not include the repartition join with Bloom filter or the zigzag join in this experiment, as even the basic repartition join is already comparable or better than broadcast join in most cases. The tradeoff between the broadcast join and the repartition join is basically broadcasting T' through the interconnection between the two clusters (the data transfered is 30×T' since we have 30 HDFS nodes) vs sending T' once through the interconnection and shuffling L' within the HDFS cluster. Due to the multi-threaded implementation described in Section 4.4, the shuffling of L' is interleaved with the reading of L in JEN, thus this shuffling overhead is somewhat masked by the reading time. As a result, broadcast join performs better only when T' is significantly smaller than L'. In our setting, broadcast join is only preferable when predicate on T is highly selective, e.g. $\sigma_T \leq 0.001$ (T' $\leq$ 25MB). In comparison, repartition-based joins are the more stable algorithms, and the zigzag join is the best HDFS-side algorithm in almost all cases.

5.2 DB-Side Joins

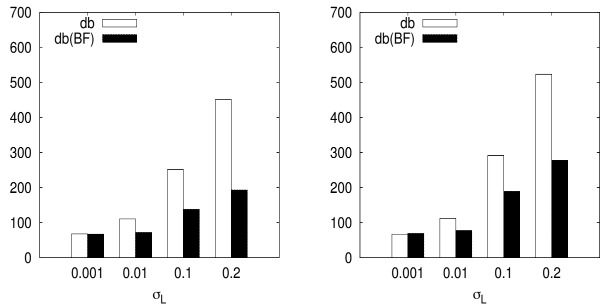
We now compare the DB-side joins with and without Bloom filter to study the effect of Bloom filter. As shown in Figure 11, Bloom filter is effective in most cases. For fixed local predicates on T (σ_T) and join-key selectivity on L' ($S_{L'}$), the benefit grows



(a) $\sigma_T = 0.001$ (b) $\sigma_T = 0.01$

Figure 10: Broadcast join vs repartition join: execution time (sec)

significantly as the size of L' increases. Especially for selective predicate on T, e.g. $\sigma_T = 0.05$, the impact of the Bloom filter is more pronounced. However, when the local predicates on L are very selective (σ_L is very small), e.g. $\sigma_L \leq 0.001$, the size of L' is already very small (e.g. less than 1GB when $\sigma_L = 0.001$), the overhead of computing, transferring and applying the Bloom filter can cancel out or even outweigh its benefit.



(a) $\sigma_T = 0.05, S_{L'} = 0.05$ (b) $\sigma_T = 0.1, S_{L'} = 0.1$

Figure 11: DB-side joins: execution time (sec)

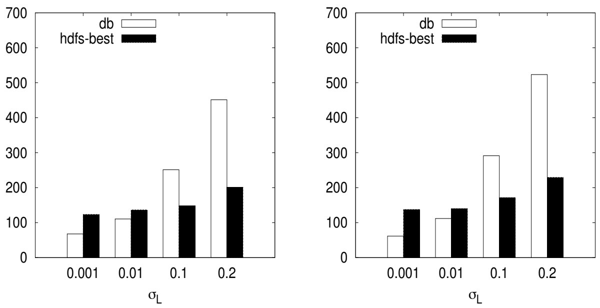
5.3 DB-Side Joins vs HDFS-Side Joins

Where to perform the final join, on the database side or the HDFS side, is a very important question that we want to address in this paper. Most existing solutions [13, 15, 14, 28] choose to always fetch the HDFS data and execute the join in the database, based on the assumption that SQL-on-Hadoop systems are slower in performing joins. Now, with the better designed join algorithms in this paper and the more sophisticated execution engine in JEN, we want to re-evaluate whether this is the right choice any more.

We start with the join algorithms without the use of Bloom filters, since the basic DB-side join is used in the existing database/HDFS hybrid systems, and the broadcast join and the basic repartition join are supported in most existing SQL-on-Hadoop systems. Figure 12 compares the DB-side join against the best of the HDFS-side joins (repartition join is the best for all cases in the figure). As shown in this figure, DB-side join performs better only when the predicate selectivity on the HDFS table is very selective ($\sigma_L \leq 0.01$). For lower selectivities, probably the common case, the repartition join shows very robust performance while the DB-side join very quickly deteriorates.

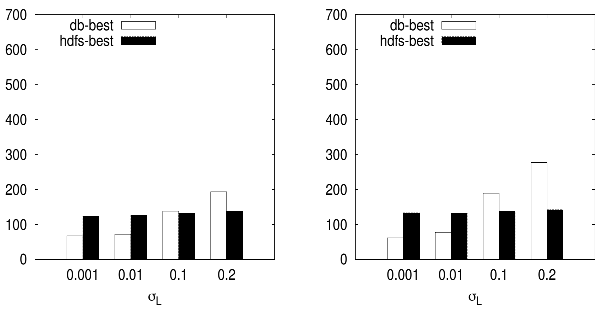
Now, let's also consider all the algorithms with Bloom filters and revisit the comparison in Figure 13. In most of the cases, the DB-side join with Bloom filter is the best DB-side join and zigzag join is the best HDFS-side join. Comparing this figure to Figure 12, the DB-side join still works better in the same cases as before, although all performance numbers are improved by the use of Bloom filters. The zigzag join shows very steady performance (execution time increases only slightly) with the increase of the L' size, in comparison with the steep deterioration rate of the DB-side join, making this HDFS-side join a very reliable choice for joins in the hybrid warehouse.

The above experimental results suggest that blindly executing joins in the database is not a good choice any more. In fact, for common cases when there is no highly selective predicate on the HDFS table, HDFS-side join is the preferred approach. There are several reasons for this. First of all, the HDFS table is usually much larger than the database table. Even with decent predicate selectivity on the HDFS table, the sheer size after predicates is still big. Second, as our implementation utilizes the DB2 optimizer as is, the HDFS data shipped to the database may need another round of data shuffling among the DB2 workers for the join. Finally, the database side normally has much less resources than the HDFS side, thus when both T' and L' are very large, HDFS-side join should be considered.



(a) $\sigma_T = 0.05$ (b) $\sigma_T = 0.1$

Figure 12: DB-side join vs HDFS-side join without Bloom filter: execution time (sec)

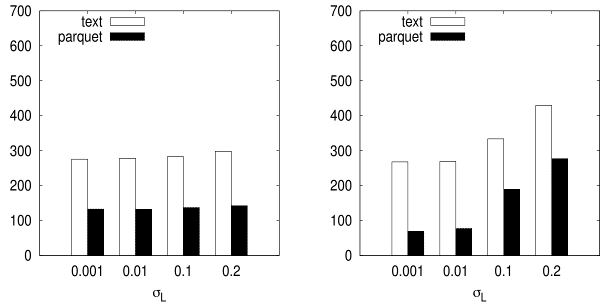


(a) $\sigma_T = 0.05$ (b) $\sigma_T = 0.1$

Figure 13: DB-side join vs HDFS-side join with Bloom filter: execution time (sec)

5.4 Parquet Format vs Text Format

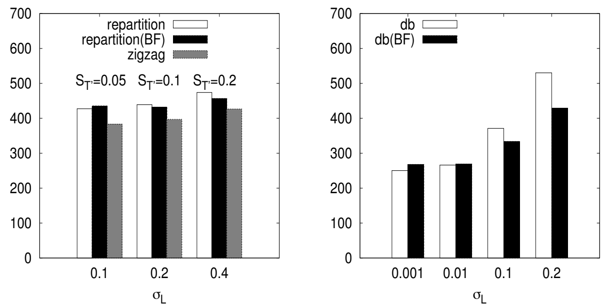
We now compare the join performance on the two different HDFS formats. We first pick the zigzag join, which is the best HDFS-side join, and the DB-side join with Bloom filter as the representatives, and show their performance on the Parquet and text formats in Figure 14.



(a) zigzag, $\sigma_T = 0.1$ (b) db(BF), $\sigma_T = 0.1$

Figure 14: Parquet format vs text format: execution time (sec)

Both algorithms run significantly faster on the Parquet format than on the text format. The 1TB text table on HDFS has already exceeded the aggregated memory size (960GB) of the HDFS cluster, thus simply scanning the data takes roughly 240 seconds in both cold and warm runs. After columnar organization and compression, the table is shrunk by about 2.4x, which can now well fit in the local file system cache on each DataNode. In addition, projection pushdown can also be applied when reading from the Parquet format. Therefore, it only takes 38 seconds to read all the required fields from the Parquet data in a warm run. This huge difference in the scanning speed explains the big gap in the performance.



(a) $\sigma_T = 0.2$ (b) $\sigma_T = 0.1$

Figure 15: Effect of Bloom filter with text format: execution time (sec)

Next, we investigate the effect of using Bloom filter in joins on the text format. As shown in Figure 15, the improvement by Bloom filter is less dramatic on the text format than on the Parquet format. In some cases of the repartition join and the DB-side join, the overhead of computing, transferring and applying the Bloom filter even outweighs the benefit it brings. Again, the less benefit of Bloom filter is mainly due to the expensive scanning cost for the text format. In addition, there is another reason for the less effectiveness of Bloom filter in the repartition join and the zigzag join. Both algorithms utilize a database Bloom filter to reduce the amount of HDFS data to be shuffled, but with multi-threading, the shuffling is interleaved with the scan of HDFS data (see Section 4.4). For text format, the reduction of the shuffling cost is largely masked by the expensive scan cost, resulting in the less shown benefit. However, for the zigzag join, with a second Bloom filter to reduce the transferred database data, its performance is always robustly better.

5.5 Discussion

We now discuss the insights from our experimental study.

Among the HDFS-side joins, broadcast join only works for very limited cases, and even when it is better, the advantage is not dramatic. Repartition-based joins are the more robust solutions for HDFS-side joins, and the zigzag join with the 2-way Bloom filters always brings in the best performance.

Bloom filter also helps the DB-side join. However, with its steep deterioration rate, the DB-side join works well only when the HDFS table after predicates and projection is relatively small, hence its advantages are also confined to limited cases. For a large HDFS table without highly selective predicates, zigzag join is the most reliable join method that works the best most of the time, as it is the only algorithm that fully utilizes the join-key predicates as well as the local predicates on both sides.

HDFS data format significantly affects the performance of a join algorithm. Columnar format with fast compression and decompression techniques brings in dramatic performance boost, compared to the naive text format. So, when data needs to be accessed repeatedly, it is worthwhile to convert text format into the more advanced format.

Finally, we would like to point out that a major contribution to the nice performance of HDFS-side joins, is our sophisticated join execution engine on HDFS. It borrows the well-known runtime optimizations from parallel databases, such as pipelining and multi-threading. With our careful design in JEN, scanning HDFS data, network communication and computation are all fully executed in parallel.

6. RELATED WORK

In this paper, we study joins in the hybrid warehouse with two fully distributed and independent query execution engines in an EDW and an HDFS cluster, respectively. Although there has been rich literature on distributed join algorithms, most of these existing works study joins in a single distributed system.

In the context of parallel databases, Mackert and Lohman defined *Bloom join*, which uses Bloom filters to filter out tuples with no matching tuples in a join and achieves better performance than semijoin [25]. Michael et al showed how to use a Bloom filter based algorithm to optimize distributed joins where the data is stored in different sites [26]. In [12], DeWitt and Gerber studied join algorithms in a multiprocessor architecture and demonstrated that Bloom filter provides dramatic improvement for various join algorithms. PERF Join [24] reduces data transmission of two-way joins based on tuple scan order instead of using Bloom filters. It passes a bitmap of positions instead of a Bloom filter of values, in the second phase of semi-join. However, unlike Bloom join, it doesn't work well in parallel settings, when there are lots of duplicated values. Recently, Polychroniou et al proposed track join [32] to minimize network traffic for distributed joins by scheduling transfers of rows on a per join key basis. Determining the desired transfer schedule for each join key, however, requires a full scan of the two tables before the join. Clearly, for systems where scan is a bottleneck, track join would suffer from this overhead.

There has also been some work on join strategies in MapReduce [8, 3, 4, 23, 44]. Changchun et al. [44] presented several strategies to build the Bloom filter for the large dataset using MapReduce, and compared Bloom join algorithms of two-way and multi-way joins.

In this paper, we also exploit Bloom filters to improve distributed joins, but in a hybrid warehouse setting. Instead of one, our zigzag join algorithm uses two Bloom filters on both sides of the join to reduce the non-joining tuples. Two-way Bloom filters require scanning one of the tables two times, or materializing the intermediate result after applying local predicates. As a result, two-way Bloom filters are not as beneficial in a single distributed system. But, in our case we exploit the asymmetry between HDFS and the database, and scan the database table twice. Since HDFS scan is a dominating cost, scanning the database table twice, especially when we can leverage indexes, does not introduce significant overhead. As a result, our zigzag join algorithm provides robust performance in many cases.

With the need of hybrid warehouses, joins across shared-nothing parallel databases and HDFS have recently received significant attention. Most of the work either simply moves the database data to HDFS [37, 38], or moves the HDFS data to the database through bulk loading [38, 17], external tables [41, 17] or connectors [18, 38]. There are many problems with these approaches. First, HDFS tables are usually pretty big, so it is not always feasible to load them into the database. Second, such bulk reading of HDFS data into the database introduces an unnecessary burden on the carefully managed EDW resources. Third, database data gets updated frequently, but HDFS still does not support updates properly. Finally, all these approaches assume that the HDFS side does not have proper SQL support that can be leveraged.

Microsoft Polybase [13], Pivotal HAWQ [15], TeraData SQL-H [14], and Oracle Big Data SQL [28] all provide on-line approaches by moving only the HDFS data required for a given query dynamically into the database. They try to leverage both systems for query processing, but only simple predicates and projections are pushed down to the HDFS side. The joins are still evaluated entirely in the database. Polybase [13] considers split query processing, but joins are performed on the Hadoop side only when both tables are stored in HDFS.

Hadapt [7] also considers split query execution between the database and Hadoop, but the setup is very different. As it only uses single-node database severs for query execution, the two tables have to be either pre-partitioned or shuffled by Hadoop using the same hash function before the corresponding partitions can be joined locally on each database.

In this paper, we show that as the data size grows it is better to execute the join on the HDFS side, as we end up moving the smaller database table to the HDFS side.

Enabling the cooperation of multiple autonomous databases for processing queries has been studied in the context of federation [21, 2, 34, 40, 30] since the late 1970s. Surveys on federated database systems are provided in [35, 22]. However, the focus has largely been on schema translation and query optimization to achieve maximum query push down into the component databases. Little attention has been paid on the actual data movement between different component databases. In fact, many federated systems still rely on JDBC or ODBC connection to move data through a single data pipe. In the era of big data, even with maximum query push down, such naive data movement mechanisms result in serious performance issues, especially when the component databases are themselves massive distributed systems. In this paper, we provide parallel data movement by fully exploiting the massive parallelism between a parallel database and a join execution engine on HDFS to speed up the data movement when performing joins in the hybrid warehouse.

7. CONCLUSION

In this paper, we investigated efficient join algorithms in the context of a hybrid warehouse, which integrates HDFS with an EDW. We showed that it is usually more beneficial to execute the joins on the HDFS side, which is contrary to the current solutions which always execute joins in the EDW. We argue that the best hybrid warehouse architecture should execute joins where the bulk of the data is. In other words, it is better to move the smaller table to the side of the bigger table, whether it is in HDFS or in the database. This hybrid warehouse architecture requires a sophisticated execution engine on the HDFS side, and similar SQL capabilities on both sides. Given the recent advancement on SQL-on-Hadoop solutions [19, 20, 33], we believe this hybrid warehouse solution is now feasible. Finally, our proposed zigzag join algorithm, which performs joins on the HDFS side, utilizing Bloom filters on both sides, is the most robust algorithm that performs well in almost all cases.

8. REFERENCES

[1] D. Abadi, P. Boncz, and S. Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., 2013.

[2] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *SIGMOD*, pages 137–146, 1996.

[3] F. Afrati and J. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[4] F. Afrati and J. Ullman. Optimizing multiway joins in a map-reduce environment. *TKDE*, 23(9):1282–1298, 2011.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[6] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, T. Westmann, I. Cetindil, and M. Cheelangi. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[7] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD*, pages 1165–1176, 2011.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[10] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB '85*, pages 151–164, 1985.

[13] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *SIGMOD*, pages 1255–1266, 2013.

[14] Dynamic access: The SQL-H feature for the latest Teradata database leverages data in Hadoop. `http://www.teradatamagazine.com/v13n02/Tech2Tech/Dynamic-Access`.

[15] Pivotal HD: HAWQ. `http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf`.

[16] HCatalog. `http://cwiki.apache.org/confluence/display/Hive/HCatalog`.

[17] High performance connectors for load and access of data from Hadoop to Oracle database. `http://www.oracle.com/technetwork/bdc/hadoop-loader/connectors-hdfs-wp-1674035.pdf`.

[18] IBM InfoSphere BigInsights. `http://pic.dhe.ibm.com/infocenter/bigins/v1r4/index.jsp`.

[19] IBM Big SQL 3.0: SQL-on-Hadoop without compromise. `http://public.dhe.ibm.com/common/ssi/ecm/en/sww14019usen/SWW14019USEN.PDF`.

[20] Impala. `http://github.com/cloudera/impala`.

[21] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: A new flavor of federated query processing for DB2. In *SIGMOD*, pages 524–532, 2002.

[22] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[23] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in MapReduce. In *RACS*, pages 100–105, 2012.

[24] Z. Li and K. A. Ross. PERF join: An alternative to two-way semijoin and bloomjoin. In *CIKM*, pages 137–144, 1995.

[25] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, page 149, 1986.

[26] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *AINA*, 2007.

[27] J. K. Mullin. Optimal semijoins for distributed database systems. *TSE*, 16(5):558–560, 1990.

[28] Oracle Big Data SQL: One fast query, all your data. `https://blogs.oracle.com/datawarehousing/entry/oracle_big_data_sql_one`.

[29] The ORC format. `http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html`.

[30] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *DOOD*, pages 161–186, 1995.

[31] Parquet. `http://parquet.io`.

[32] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[33] Presto. `http://prestodb.io`.

[34] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, pages 664–682. ACM Press/Addison-Wesley Publishing Co., 1995.

[35] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[36] Snappy. `http://code.google.com/p/snappy`.

[37] Sqoop. `http://sqoop.apache.org`.

[38] Teradata connector for Hadoop. `http://developer.teradata.com/connectivity/articles/teradata-connector-for-hadoop-now-available`.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[40] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *TKDE*, 10(5):808–823, 1998.

[41] Teaching the elephant new tricks. `http://www.vertica.com/2012/07/05/teaching-the-elephant-new-tricks`.

[42] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[44] C. Zhang, L. Wu, and J. Li. Optimizing distributed joins with bloom filters using MapReduce. In *Computer Applications for Graphics, Grid Computing, and Industrial Environment*, pages 88–95, 2012.