K.-U. Sattler et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2021),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 59

Optimized Theta-Join Processing through Candidate Pruning and Workload Distribution

Julian Weise[1], Sebastian Schmidl[2], Thorsten Papenbrock[3]

Abstract: The Theta-Join is a powerful operation to connect tuples of different relational tables based on arbitrary conditions. The operation is a fundamental requirement for many data-driven use cases, such as data cleaning, consistency checking, and hypothesis testing. However, processing theta-joins without equality predicates is an expensive operation, because basically all database management systems (DBMSs) translate theta-joins into a Cartesian product with a post-filter for non-matching tuple pairs. This seems to be necessary, because most join optimization techniques, such as indexing, hashing, bloom-filters, or sorting, do not work for theta-joins with combinations of inequality predicates based on $<, \leq, \neq, \geq, >$.
In this paper, we therefore study and evaluate optimization approaches for the efficient execution of theta-joins. More specifically, we propose a theta-join algorithm that exploits the high selectivity of theta-joins to prune most join candidates early; the algorithm also parallelizes and distributes the processing (over CPU cores and compute nodes, respectively) for scalable query processing. The algorithm is baked into our distributed in-memory database system prototype A^{2}DB. Our evaluation on various real-world and synthetic datasets shows that A^{2}DB significantly outperforms existing single-machine DBMSs including PostgreSQL and distributed data processing systems, such as Apache SparkSQL, in processing highly selective theta-join queries.

Keywords: theta-join; query optimization; distributed computing; actor programming

1 Theta-Join Processing

A *join* is a powerful operation in relational database theory that allows us to combine tuples of the same or different relational instances. The most popular join operator is the *equi-join* $\bowtie$ that combines tuples based on the equality of certain attribute values. The equi-join serves most basic tuple combination scenarios, such as tuple reconstruction in normalized schemata, knowledge enrichment via data integration, and the resolution of foreign-key relationships. The *theta-join* $\bowtie_\Theta$ is a generalized join variant that combines tuples based on arbitrary join conditions Θ including but not limited to value equality. A join condition is a boolean statement on the attribute values of two tuples. Following related work, we express any such statement as a conjunction of predicates based on $<, \leq, \neq, \geq, >$.

[1] Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany julian.weise@hpi-alumni.de
[2] Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany sebastian.schmidl@hpi.de
[3] Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany thorsten.papenbrock@hpi.de

doi:10.18420/btw2021-03

Geo-spacial querying. Theta-joins are used whenever tuples need to be paired up in a specific way. In *temporal* or *geo-spacial querying*, for example, tuples often need to match in certain value ranges. The query "Combine all *City*-tuples with those *State*-tuples in which they are geographically located"(see Figure 1), for instance, matches tuples based on longitude (*Long*) and latitude (*Lat*) range information.

$$\rho_C(City) \bowtie_{S.Long_{min} \leq C.Long \wedge S.Long_{max} \geq C.Long \wedge S.Lat_{min} \leq C.Lat \wedge S.Lat_{max} \geq C.Lat} \rho_S(State)$$

Fig. 1: A geo-spatial query that builds tuples of cities and their corresponding states.

Data cleaning. Another important area of application for theta-joins is rule-based *data cleaning*. Given a data quality rule or an integrity constraint, such as "*TaxPayer*s with higher income need to pay more taxes than *TaxPayer*s with lower income", we can formulate the negated constraint as a theta-join query (see Figure 2) to retrieve all data inconsistencies w. r. t. this constraint from the data and clean them afterwards.

$$\rho_{T1}(TaxPayer) \bowtie_{T1.Income < T2.Income \wedge T1.TaxRate > T2.TaxRate} \rho_{T2}(TaxPayer)$$

Fig. 2: A data cleaning query retrieving all inconsistent tuple pairs w. r. t. a given integrity constraint.

Hypothetis testing. The use of theta-joins in the area of *hypothetis testing* works very similar to the data cleaning use case: Given a hypothethis statement, such as "*Countries* that invest more in education have less child poverty and a higher educational level than *Countries* that invest less in education", we can query all pairs of countries that contradict this statement via a theta-join (see Figure 3).

$$\rho_A(Country) \bowtie_{A.ESpend > B.ESpend \wedge (A.CPov \geq B.CPov \vee A.ELevel \leq B.ELevel)} \rho_B(Country)$$

Fig. 3: A hypothesis testing query that collects contradicting tuple pairs.

Hypothesis and integrity statements are often formulated manually by domain experts. They can, however, also be discovered automatically with modern *data mining* and *data profiling* algorithms. Functional dependencies, order dependencies and denial constraints are only a few types of statements that can meanwhile be retrieved automatically [AGN15]. While it has been shown that the mined statements are useful for tasks, such as consistency checking and data cleaning [Bo07; Co17], the amount and complexity of the statements puts significant pressure onto the theta-join operations used for their evaluation.

Although the theta-join is an essential part of relational algebra, its common physical implementation in data query engines is a nested-loop join, i. e., the Cartesian product, in combination with a filter operation. Consider for example the theta-join shown in SQL Query 1, which targets the San Francisco `Employee Compensation` dataset[4].

[4] `https://data.sfgov.org/City-Management-and-Ethics/Employee-Compensation/88g8-5mnd` (08-August-2020)

```sql
SELECT a.eID, b.eID
FROM EmployeeCompensation a, EmployeeCompensation b
WHERE a.jobCode = b.jobCode
    AND a.salaries < b.salaries
    AND a.eID != b.eID
```

SQL Query 1: Identify pairs of employees performing the same job but being paid differently.

We executed the query on different query engines including *PostgreSQL*[5], *SparkSQL*[6] and *Amazon Redshift*[7], which all produced a query plan similar to the one shown in Listing 2: an equi-join on the equality predicates followed by a post-filter on the inequality predicates.

```
Merge Join
    Merge Cond: (a.jobcode = b.jobcode)
    Join Filter: ((a.salaries < b.salaries) AND (a.ID <> b.ID))
    -> Sort: Sort Key: a.jobcode
        ->  Seq Scan on EmployeeCompensation a
    -> Materialize
        ->  Sort: Sort Key: b.jobcode
            ->  Seq Scan on EmployeeCompensation b
```

Listing 2: The PostgreSQL query plan for SQL Query 1.

With the high selectivity of the equi-join, the query offers a relatively good performance. However, considering the queries of the use cases discussed above, most of them do not employ equality operators. So replacing the equality operator in Query 1 with an inequality operator causes the query engines to produce queries plans similar to the one shown in Listing 3: a nested-loop join with a large post-filter.

```
Nested Loop
   Join Filter: ((a.jobcode <> b.jobcode)
   AND (a.salaries < b.salaries) AND (a.ID <> b.ID))
   ->  Seq Scan on EmployeeCompensation a
   ->  Materialize
       ->  Seq Scan on EmployeeCompensation b
```

Listing 3: PostgreSQL Query Plan for SQL Query 1 without equality operator

Because the established hashing-, indexing- and sorting-based optimizations are not applicable for complex join conditions with inequality predicates, the systems fall back to the quadratic comparison of all tuples without employing any optimization. The performance of the nested-loop join in all systems is, therefore, dramatically worse than the performance

[5] https://www.postgresql.org/ (08-August-2020)
[6] https://spark.apache.org/sql/ (08-August-2020)
[7] https://aws.amazon.com/redshift/ (08-August-2020)

of an equi-join. This is problematic when executing theta-joins on real-world datasets as the costs scale quadratically with the datasets' size.

Although there is probably no solution for the quadratic complexity of theta-joins, we can still optimize the join performance by *pruning join candidates* (and, hence, their join condition tests) and by *distributing the join workload* to multiple machines (and, hence, scale out the processing). In this paper, we develop a theta-join algorithm that implements these two optimizations for our distributed in-memory database system prototype A^2DB. A^2DB is an actor-based and, therefore, inherently parallel and distributable database, which is designed for analytical query workloads. It builds upon the *actor model*, which is a reactive programming paradigm that uses actors as its universal computational primitives. An actor is essentially an object with strictly private state that communicates with other actors using asynchronous messaging. The architecture of this system follows the idea of an actor database system [Be18; SSP19], in which all database state is encapsulated in actors. Our database prototype and, hence, also our theta-join algorithm are implemented using the *akka toolkit*[8], which is the most popular actor model implementation for the Java Virtual Machine.

Join candidates pruning. Our first optimization is based on the observation that theta-join results in real-world use cases are small (often even empty) and grow rather linearly with the size of the data: Geo-spacial queries result in manageable overlaps, data cleaning queries should return relatively few data quality issues, and hypothesis checking queries are expected to return empty results (or very small results if the hypothesis is not quite correct). For this reason, most real-world theta-joins have a high selectivity. In this paper, we propose a theta-join algorithm that calculates and evaluates the selectivity of the individual join predicates; selective predicates are, then, used to prune the candidate space.

Join workload distribution. Because theta-join results have a quadratic worst-case size in the length of the input dataset, the candidate pruning effects are not always sufficient to process larger join queries. For this reason, our theta-join algorithm facilitates parallelization and can be scaled out to multiple compute nodes. The ability to scale also naturally exploits the distributed storage of data in the A^2DB system.

In the following, we first discuss related work in the area of (theta-)join processing and the limitations of existing approaches (Section 2). We then explain how data is maintained and distributed (Section 3). With these basic details explained, we first describe our distributed theta-join algorithm (Section 4) and then its selectivity-based join strategies (Section 5). In an extensive evaluation, we then compare the performance of our theta-join algorithm with the performance of the data processing systems *PostgreSQL* (single node), *SparkSQL* (12 node cluster) and *Amazon Redshift* (12 node cloud) to demonstrate that A^2DB can process selective theta-joins significantly faster than the state-of-the-art Carthesian product plus post-filter approach (Section 6).

[8] `https://akka.io` (08-August-2020)

2 Related Work

In this section, we give an overview of existing work in the area of theta-join processing. We take a brief look at the origin of join operations in the relational model but focus our investigation on efficient and distributed answering of theta-joins.

Relational Joins were first discussed by Codd in 1970 as a concept of combining tuples based on attribute equality in his proposal for the relational data model. He later extended his proposal by combining tuples also with non-equality operators, which was the introduction of the *theta-join* operator [Co79]. Early subsequent work then mainly focused on the equality-based join operation and suggested various implementations and optimizations to calculate these equi-joins efficiently [Go75]. Prominent examples are hash, sort-merge, and nested-loop joins, as well as techniques utilizing indexes [ME92].

Parallel and Distributed Equi-Join Processing techniques have been examined extensively in response to the development of multi-core machines. Specifically, researchers identified challenges and proposed solutions for typical problems in multi-threaded and distributed systems, such as shared state and workload partitioning [ESW78; VG84]. A prominent optimization for calculating joins in distributed systems, which was proposed by Bernstein, utilizes semi-joins to extract join candidates [BC81]. In this way, the communication overhead and, hence, query processing time could be decreased significantly. Most of the techniques proposed in this area can, however, not be applied to theta-joins, because they do not support complex theta predicates.

Efficient Theta-Join Processing is the goal of the *IE-Join* algorithm by Khayyat et al. [Kh15]. The algorithm applies a sophisticated sort-merge approach by first sorting the values of up to two join attributes and implicitly identifying candidate sets. Via permutation arrays and clever bitset operations, the algorithm tests all predicates of the join condition successively while effectively pruning candidates on the way. In this way, IE-Join is orders of magnitude faster than both PostgreSQL and SparkSQL. Despite its superior performance, the proposed approach is inherently limited to only two join predicates. Adapting IE-Join to more than two predicates requires exactly the strategies proposed in this paper: a strategy to choose the two IE-Join predicates and a post-processing step for all non-chosen predicates.

Distributed Theta-Join Processing optimizations mostly target batch processing and data flow engines, such as *Apache MapReduce* or *Apache Spark*. These engines are effective in processing equi-joins, because distributed grouping and aggregation of tuples is baked into their core feature set, but ineffective for theta-joins, because the grouping does not innately support inequality operators. The *1-Bucket-Theta* algorithm by Okcan et al. [Ok11] is a theta-join processing approach on MapReduce that splits the quadratic comparison space into buckets, which are then processed distributedly by different machines to share the comparison load. *M-Bucket-Theta* enhances the 1-Bucket-Theta algorithm in that the algorithm can detect empty regions in the matrix and prune non-contributing join candidates. Because the algorithm depends on a single comparison matrix, theta-joins with more than

one predicate are handled by concatenating the predicates' attributes into one key. For this reason, M-Bucket-Theta can handle theta-joins with only one predicate effectively. The work of Koumarelas et al. [KNG18] proposes several strategies to optimize M-Bucket-Theta's efficiency for low-selectivity queries. By manipulating the matrix of the mapping phase such that larger regions of it can be pruned, the strategies reduce the algorithm's communication and computation costs by up to 45% and 50% respectively. Despite these performance improvements, the theta-join algorithm still cannot handle multiple predicates. Because multiple-predicate theta-joins are a given for most use cases, such as hypothesis testing and data cleansing, we do not evaluate these approaches in this work.
To join more than two relations at once, Zhang et al. studied the problem of decomposing a multi-way theta-join into multiple binary joins and proposed different strategies and a cost model for optimizing the overall processing time [ZCW12]. For the execution of chained joins, the authors rely on variations of M-Bucket-Theta. In this paper, we consider multi-way theta joins as orthogonal work and focus on efficiently joining two relations.
Besides the MapReduce-based theta-join approaches, Apache Spark supports join processing with arbitrary join conditions innately with its relational module *SparkSQL* [Ar15]. The data flow engine offers *DataFrames* as an abstraction for distributed datasets and an SQL engine to query these datasets. Although the engine also falls back on *Broadcast-Nested-Loop-Join*s when processing theta-joins, the framework is significantly faster than MapReduce.
The capability of distributed theta-join processing can also be found in many commercial DBMSs, such as *Amazon Redshift*. Redshift is a distributed data warehouse solution hosted exclusively in the Amazon Web Services (AWS) cloud. It distributes data across a cluster of configurable size and involves all nodes in query answering. Redshift in particular claims itself to be an efficient and scalable solution for experimenting with (possibly huge) amounts of data [Gu15], which makes it a perfect baseline for our experimental evaluations.

3 Data storage in A^{2}DB

Before we introduce our theta-join algorithm, we need to explain how our database system prototype A^{2}DB stores and handles data. A^{2}DB is an actor-based, distributed in-memory relational DBMS for analytical query workloads that facilitates a leader-follower architecture:

Leader Node: One dedicated node in the A^{2}DB cluster takes the role of a leader. It is responsible for bookkeeping the follower node's membership state, accepting queries and loading data into the database.

Follower node: An A^{2}DB follower is a node in the cluster that is responsible for maintaining and querying portions of the data. Which portions, i. e., partitions of the data a follower is responsible for is defined by the leader node.

Follower nodes play an active role in query processing: The leader node breaks every submitted query, such as a theta-join, into multiple work packages and assigns them to individual follower nodes. Once a follower node receives a work package, it requests

necessary data from other nodes, decides the best local query execution strategy, processes the local results, and sends the results of the query back to the master.

The partitioning of the data in A^2DB follows the *PAX* concept: The entire relational dataset is sliced horizontally into equally sized partitions and every partition is stored columnar-wise on one follower node. For every column in a partition, A^2DB maintains column-specific metadata, such as the column's *min* and *max* values in this partition and a pre-calculated *sorting* of the partition tuples w. r. t. this column. During query processing, the extreme values can be used to prune this partition and the pre-sortings support sort-based query operators, such as sort-merge joins.

Strictly following the actor programming model, all partitions in A^2DB are represented as autonomous actors, which is, in private, non-parallelizable actor state. To access data owned by another actor (e. g., in a join scenario), partition holder actors need to ask other partition holder actors via asynchronous messaging for certain tuples, columns, or values.

4 Theta-Join Workload Distribution

When theta-joining two relations R and S, the query engine needs to validate each possible combination of tuples from both relations against the join condition Θ. Hence, up to $|R| \times |S|$ comparisons need to be performed. Our first approach to efficiently process these comparisons is to distribute the workload to any given number of nodes. When a query is issued, the data is already horizontally partitioned on these nodes. In this section, we propose a reactive join strategy that decomposes and distributes the Θ evaluations. Figure 4 visualizes the general idea of our approach with an example: A^2DB splits the join space of two relations R and S and their partitions Ri and Si into *node-joins*, such as $R_A \bowtie_\Theta S_B$, and each node-join then into *partition-joins*, such as $P_{R1} \bowtie_\Theta S_{S1}$. A partition-join comprises two partitions P_{Ri} and P_{Si} and constitutes the smallest work-package in the system. In the example, the theta-join consists of four node-joins and each node-join consists of four partition-joins – usually, though, an A^2DB cluster consists of more nodes and partitions.

Figure 5 depicts the process of executing a theta-join: When the leader node receives a theta-join query, it creates the node-join matrix that partitions the join into node-joins. It then opens a query session, which causes all follower nodes to calculate their local node-join, i. e., all $R_A \bowtie_\Theta S_A$, $R_B \bowtie_\Theta S_B$ etc. To perform a partition-join, a processor evaluates all tuple combinations $((t_R, t_S) \mid t_R \in P_{R_i}, t_S \in P_{S_j})$ against the join condition Θ and sends the matching tuples to the result set of the theta-join on the leader. Whenever a follower node finishes a node-join, the leader serves the follower with another node-join. This reactive work pulling mechanism keeps all cluster nodes busy until the join is completed. The leader coordinates this process so that, in the end, all partition-joins are executed. Later in this section, we discuss the leader's node-join selection strategy in more detail.

Every node-join is calculated on one follower node. On that node, the calculation is strongly parallelized and consists of three overlapping steps, which are also shown in Figure 5: The

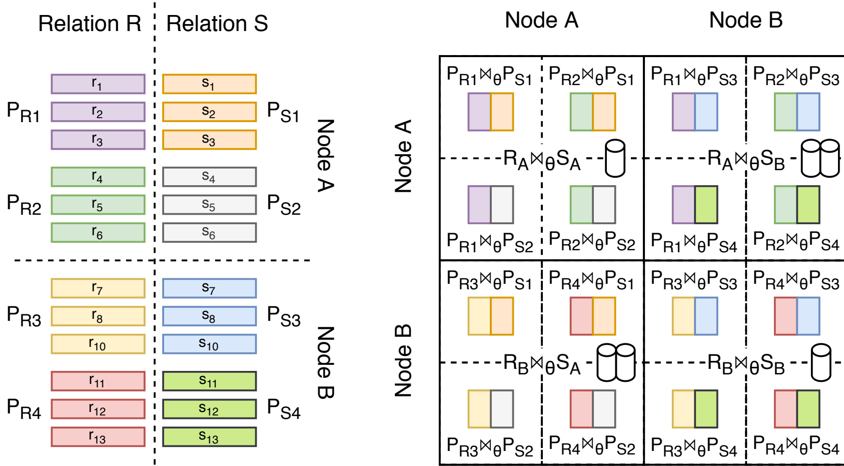


Fig. 4: Distribution of the theta-join calculations over two nodes.

work generation step splits the node-join into partition-join tasks. The *data loading* step then fetches all necessary remote partition data on demand from the other node of the current node-join; the step makes sure that every partition is retrieved only once and it is skipped by the self node-joins, e. g., $R_A \bowtie_\Theta S_A$. Once a remote partition is available, the *execution step* can start to join this partition with every local partition; every partition-join is executed reactively on one actor and, hence, in parallel to other partition joins. Once all partition-joins are calculated and their results are send to the leader, the node-join is completed and the follower is ready for the next node-join.

In the following, we discuss the orchestration of the node-joins (Section 4.1), the provisioning of partitions (Section 4.2), and the actual join processing (Section 5).

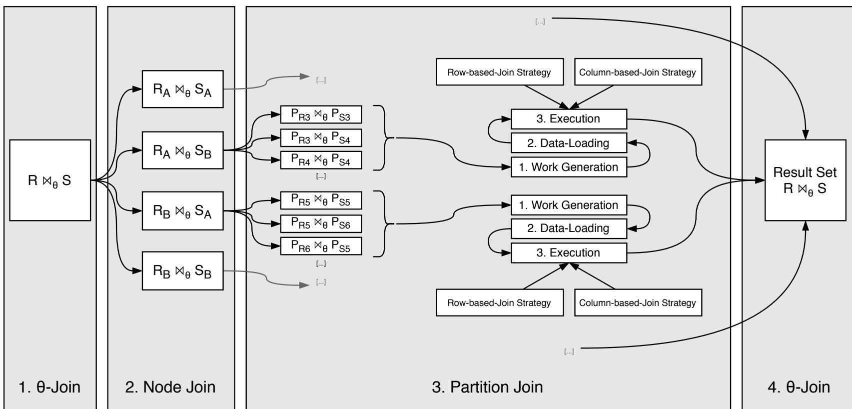


Fig. 5: Overview of the entire distributed theta-join processing process.

4.1 Execution Plan

The *execution plan* is essentially a dynamically created actor on the leader that represents the distributed execution of a theta-join query. It opens a cluster-wide session (involving session actors on all follower nodes), orchestrates the intermediate query processing steps and collects the query result. Through the session actors, all follower nodes know each other and can communicate on a peer-to-peer basis.

With the execution plan, A^{2}DB aims to orchestrate the node-joins in an optimal way. It does so by sending out node-joins as work packages to the followers' query *executors*. The initial task for each follower's executor is to calculate the local node-join. Afterwards, the executors start pulling further node-joins from the execution plan and the task of the execution plan is to serve these requests in a best possible way. This means primarily that every node-join should be handled by a node that owns at least one of the node-join's sides, i. e., $R_A \bowtie_\Theta S_B$ should be handled by node A or B. Furthermore, we assign both node-join directions to the same node, i. e., $R_A \bowtie_\Theta S_B$ and $S_B \bowtie_\Theta R_A$ are one work package that goes to either node A or node B. In this way, partitions are not send in both directions. However, by assigning the node-joins naively in, for instance, node order, the execution plan quickly encounters requests by a node, whose partitions have all already been joined elsewhere, and therefore cannot serve this node with optimal work. Because followers usually finish their node-joins unevenly fast, the execution plan cannot plan the node-join distribution in advance and, instead, chooses the node-joins reactively based on three heuristics:

1. **Data Locality**: Assign a node-join that involves the partitions of the requesting node, if possible. This rule has the highest priority and overrules all other heuristics.
2. **Selection Flexibility**: Assign a node-join with the least often joined node. By joining the least often joined node next, the execution plan maintains the highest possible flexibility for future join selections – it effectively tries to avoid situations where all node-joins of a particular requesting node are already done. For this, the execution plan counts the number of already assigned node-joins per node.
3. **Query Politeness**: Assign a node-join with the least often requested node. If all potential join partner have the same join counts, selecting the least often requested partner should avoid uneven loads for sending out local partitions. For this heuristic, the execution plan also counts the number of partition requests per node.

Work assisting: Despite these heuristics, the execution plan cannot always meet the first rule, especially at the end of the execution. So if a node cannot be served with a node-join involving itself, the execution plan assigns a node-join according to rule two and three. We refer to the process of taking over foreign node-joins as *work assisting*. The processing of such node-joins requires the execution node to fetch partitions from two nodes instead of one, which is more expensive. Hence, to decide whether work assisting is actually beneficial, the execution plan tracks three additional runtime metrics per node: the average execution time of node-joins $t_{average}$, the execution time for the current node-join $t_{elapsed}$, and the average partition transfer delay t_{delay}. Then, work stealing is done only if the expected

remaining execution time ($t_{average} - t_{elapsed}$) is larger than the expected additional network delay of executing the next node-join as a foreign-node-join (t_{delay}).

Work stealing: If work assisting is no longer possible, follower nodes may support other follower nodes in finalizing their current node-joins by "stealing" some partition-joins. In contrast to work assisting, *work stealing* does not take over an entire node-join but some portion of the remaining partition-joins. For this, the execution plan actor instructs the work requesting follower to steal half of the partition-joins from the follower with the shortest current node-join time $t_{elapsed}$, which should be the follower with the heuristically most unfinished partition-joins. The stealing follower than retrieves these partition-join tasks from the target follower in a peer-to-peer fashion. Both followers report their join results directly to the leader; the leader takes care that no follower is "robbed" more than once.

4.2 Context-specific Partition Provisioning

Before a follower node requests partitions from another follower node, it first exchanges both the join condition Θ and the headers of the involved partitions (see Section 3) with the other follower. The join condition and header metadata help the follower nodes to exchange only required, i. e., context specific partition data. Given the node-join $R_A \bowtie_\theta S_B$, then only a portion of S's partitions on node B are relevant for the node-join on node A:

1. A requires only those attribute values from B's partitions that are used in Θ. Thanks to the column-oriented format of the partitions, these attributes can easily be selected.
2. A requires only those records from B's partitions that intersect with A's partitions w. r. t. all of Θ's attribute-specific join operators, which are $<, \leq, =, \neq, \geq, >$. The overlap can be checked quickly with the partition's min and max values of each attribute.

As an example for condition 2, if two partitions P_{Ri} and P_{Sj} have no overlap in attribute x, i. e., $P_{Ri}.x_{min} > P_{Sj}.x_{max}$ and the join condition is $R.x < S.x$ or $R.x \leq S.x$, then A^{2}DB does not transmit P_{Sj}, because the join of these partitions is empty. If P_{Ri} and P_{Sj} overlap partially, the records are filtered so that the range (min and max) of the transmitted values matches all Θ conditions. In other words, given $R.x < S.x$, node B sends only those local P_{Sj} records where $P_{Sj}.x > P_{Ri}.x_{min}$. With this minimal checking overhead, A^{2}DB can prune many partition values from the sending process.

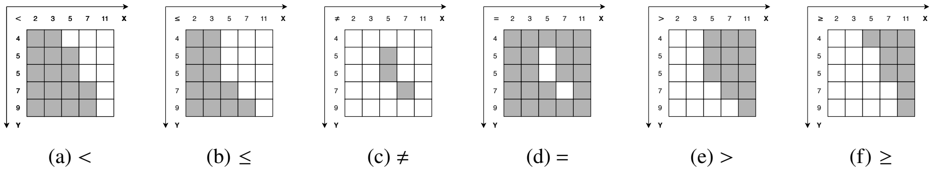
5 Theta-Join Candidate Pruning

As already shown in Figure 5, the node-join processing consist of three steps: work generation, data loading, and execution. The work generation splits the node-join into partition-joins and puts the resulting tasks into a task queue. To not overload the memory or network

and to allow other followers to steal work from the task queue, data loading and execution operate on a pull-based execution model: Free worker actors consume partition-join tasks, which first causes missing partition data to be loaded and, once the data arrives, be joined. Via slight over-provisioning and data pre-fetching, the A^{2}DB follower nodes maximize both CPU and network utilization. The final partition-join execution step takes as input the partitions P_{Ri} and P_{Sj}, the partition header metadata, and the join condition Θ.

5.1 Predicate-specific Selectivity Calculation

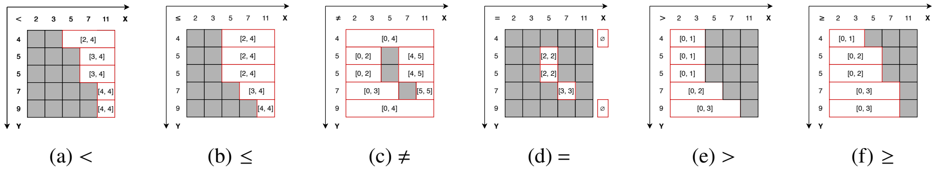
Before A^{2}DB starts the actual join calculation, it first determines the selectivity of each join predicate $P_{Ri}.x \bowtie_{\vartheta} P_{Sj}.x$ in Θ with $\vartheta \in \{<, \leq, =, \neq, \geq, >\}$. Based on the selectivities, each follower can later choose the best join strategy for its current partition-join. To calculate the selectivity for each join predicate, A^{2}DB exploits the pre-calculated sortings of every attribute in a way that requires at most $2 \cdot (|P_{Ri}| + |P_{Sj}|)$ tuple comparisons. For this, we interpret each predicate as a $|P_{Ri}| \times |P_{Sj}|$ matrix of record pairs. We then draw two lines into this matrix that separate matching tuples from non-matching tuples w. r. t. the predicate's join operator ϑ. Figure 6 shows example results for all operators. To calculate the selectivity, we simply sum up all ranges of matching tuples and divide the result by $|P_{Ri}| \cdot |P_{Sj}|$.



(a) $<$ (b) $\leq$ (c) $\neq$ (d) $=$ (e) $>$ (f) $\geq$

Fig. 6: Exemplified join matrices for all join operators supported by A^{2}DB

To calculate the border lines efficiently, a partition-join worker starts with two pointers l and r in the left-upper corner of the matrix. It then compares the records at the pointer locations and advances the l pointer in a way that it follows the left index of matching tuples and r follows the right index of matching tuples. So for example, if $P_{Ri}(l)\ \vartheta\ P_{Sj}(l)$ is true, i. e., the records' values at pointer location l match, l advances downwards; otherwise, it advances to the right. For the same comparison, r advances to the right for matches and downwards, otherwise. The calculation ends when both pointers arrive at the bottom of the matrix. Note that this procedure does not work for $\neq$, because $\neq$ defines two areas of matching tuples; hence, A^{2}DB calculates $\neq$ as $=$ and inverts the resulting counts. The matching tuple pairs are technically stored in an $|P_{Ri}|$ long array of ranges (see Figure 7). We call this the join *candidate set* for predicate ϑ. The range indexes are given by the l and r pointers whenever these pointers move downwards. The selectivity calculation is executed for all join predicates of Θ in parallel and finishes when all branches are done.



(a) < (b) ≤ (c) ≠ (d) = (e) > (f) ≥

Fig. 7: Representation of matching tuples pairs in ranges

5.2 Partition-Join Strategies

The intersection of all candidate sets, i. e., all tuple pair sets of all predicates, is the theta-join result for $P_{Ri} \bowtie_{\Theta} P_{Sj}$. However, because the join matrices (mostly) use different sortings, the calculated ranges cannot be intersected directly. There are basically two join strategies that we can follow at this point: *row-oriented joining* and *column-oriented joining*.

Row-oriented Join Strategy: The row-oriented joining strategy takes a set of join candidates, which is, for example, the candidate set of one join predicate, and checks each candidate against the entire join condition Θ. This immediately validates the entire join tuple, i. e., one result row. Because Θ is formulated in conjunctive normal form, the predicate testing for one tuple pair stops immediately and discards the pair if a predicate is invalid.

Column-oriented Join Strategy: The column-oriented joining strategy successively intersects the sets of matching tuple pairs of all join predicates. Thereby, the strategy basically tests one predicate after the other vertically for all result tuples, i. e., column-wise for each join attribute. To intersect two candidate sets, which are represented as lists of tuple ranges based on attribute-specific sortings, A^{2}DB first translates both candidate sets into matrices with same tuple sortings in both dimensions. Both resulting matrices are represented as $|P_{Ri}|$-long array of sparse bitsets (1-bits for matches; 0-bits for no-matches). After this transformation, A^{2}DB can efficiently intersect these candidate sets. The costs for the transformation and intersection depend on the selectivities of the two join predicates.

Both the row- and the column-oriented join strategy profit from considering the most selective predicates, which is the one with the lowest selectivity factor, first: The candidate set of the most selective predicate is the smallest and, hence, prunes the most candidate evaluations when chosen as initial candidate set for row-oriented joining; similarly, intersecting the sparsest candidate sets first in column-oriented joining maximizes the pruning effect of every intersection step and, hence, the overall compute efficiency. For this reason, A^{2}DB sorts the predicates by their selectivity factors and uses the most selective predicates first – regardless of the join strategy, which it needs to choose right after sorting the predicates.

Choosing the row-oriented strategy effectively uses one candidate set for pruning. However, if there is no single predicate with a high selectivity (low selectivity factor) and further predicates are needed to sufficiently prune the candidate space, this strategy alone looses a lot

of pruning potential. On the other hand, choosing the column-oriented strategy exploits all pruning aspects, but because the transformation of range-based into bitset-based candidate sets is expensive, translating all candidate sets outweighs the pruning effect. Considering the examples in Figure 6 shows that, for instance, neither (b) nor (f) are particular effective, but their intersection, which is (d), is highly selective; the candidate set (c) is neither alone nor in any combination effective enough to compensate its translation costs. For these reasons, we propose a *strategy selection heuristic* that combines both approaches.

Strategy Selection Heuristic: All workers decide the join strategy for their current partition-join task based on the local selectivities and independently of one another. After all predicates are sorted in ascending order according to their selectivity factors, a worker follows the following decision heuristic:

1. **Only one predicate**: If the join condition Θ consists of only one predicate, A^2DB simply returns the candidate set of that one predicate, which is the result of the current partition-join.
2. **100% selective**: If the most selective predicate matches no tuple pair, the partition-join result is empty, because the predicate with the highest selectivity defines an upper bound for the number of join results; hence, an empty set is returned.
3. **95 – 100% selective**: If the most selective predicate matches only at most 5%, it alone is so selective that column-oriented joining is not promising. For this reason, the worker uses only the first predicate's candidate set as input for row-oriented joining.
4. **< 95% selective**: If no highly selective predicate exists, the worker intersects the two most selective predicates via column-oriented joining and feats the resulting candidates into row-oriented joining.

The last case uses only the first two predicates for column-oriented joining, because we observed that the first two predicates usually have such a strong combined pruning effect that adding a third predicate does not pay off. Different settings of the proposed 95% decision threshold can cause faster execution times depending on various factors, such as dataset size, cluster size and cluster speed, but thresholds around 95% showed the best and also very similar (hence robust) performance results in our experiments. This is due to the generally high selectivity of most real-world theta-join queries and the fact that all workers choose their strategies independently.

Whenever a worker finishes a partition-join, it sends the results to the leader node and fetches the next partition-join from the work queue. After completing all partition-joins, the current node-join is completed and the follower pulls the next node-join from the leader's execution plan. Once all node-joins are done, the execution plan actor reports the final result to the query issuing client and closes the join session.

6 Evaluation

We now evaluate A^2DB's theta-join performance against the theta-join performance of three state-of-the-art data query engines: *PostgreSQL* as a modern representative for a single machine (non-distributed) DBMS, *Amazon Redshift* as a distributed and highly scalable relational DBMS, and *Apache SparkSQL* as a distributed batch-processing system with theta-join capabilities. For the experiments, we configured these systems as follows:

A^2DB runs on a 12 node cluster, where each node has an Intel Xeon E5-2630 v4 CPU (20 threads), 32 GiB RAM and 1 GiBit/s Ethernet. The nodes run Ubuntu 18.04.4 and Java 1.8 with G1 garbage collector. A^2DB uses a maximum partition size of 5,000 tuples for datatses smaller than 500,000 tuples and a maximum partition size of 10,000 tuples, otherwise. In this way, each node hosts at least one maxed out partition in all evaluations.

PostgreSQL version 10.12 uses one of the nodes described above, but with 64 GiB RAM. We optimized the default configuration of PostgreSQL to achieve a better performance for analytical workloads as suggested in the official documentation[9]: We set the `shared_buffers` to 25% of the system's main memory, which is 16 GiB. The `work_mem` is increased to 512 MB, to not exhaust the memory but still provide enough memory for executing queries mainly in memory. We also increased the `temp_buffers` to 512 MB.

Apache SparkSQL version 2.4.4. uses the same cluster than A^2DB. The driver program is written in scala version 2.12 and uses the Hadoop distributed file system as storage technology for the datasets and the query results.

Amazon Redshift needs to be hosted on different hardware in the AWS cloud. Its cluster consists of 12 dc2.large nodes and runs Redshift version 1.0.17498. Each node is an EC2 cloud computing resource with an Intel E5-2686 v4 CPU (two threads), 15 GiB RAM, and 160 GiB NVMe SSDs. To run only one query at a time, we changed the query queue configuration to prohibit concurrent query execution and use all available memory.

Because the hardware for Redshift differs, we define the following rules for comparing the query times of the four systems: A^2DB is truly better than PostgreSQL only if it is at least 11 times faster than PostgreSQL (because it has 11 times more nodes); A^2DB is better than Redshift only if it is at least 10 times faster (because it has 10 times more threads) and A^2DB is clearly slower if Redshift is faster despite its disadvantage – otherwise, we cannot specify which query processing time is better as we do not know Redshift's scalabilty with the number of local threads (note that Redshift is also highly optimized and specifically tuned for being executed on AWS hardware); A^2DB and SparkSQL are directly comparable.

For our experiments, we use synthetic and real-world datasets, which are differently sized subsets of four base recordsets (see Table 1): The synthetic *TPC-H* benchmark dataset, the employee compensation dataset *DataSF* of San Francisco, the US Bureau of Transportation

[9] `https://www.postgresql.org/docs/10/runtime-config-resource.html` (08-August-2020)

dataset *Flight*, and the extended edited synoptic *Cloud* reports dataset. For the purpose of identifying single rows, we extended all datasets with an additional surrogate key, which is a dedicated, monotonic increasing integer *id* column. We cut down the *Cloud* dataset to five million records, because all systems struggled with its entire size.

Dataset	# Rows	# Columns	Size on disk	Domain	Real-World
TPC-H	6,001,215	25	1,639 MB	Order Management	✗
DataSF	968,373	22	197 MB	Public Administration	✓
Flight	7,268,232	15	701 MB	Flight Control	✓
Cloud	384,584,555	28	521 MB	Weather	✓

Tab. 1: Recordsets used for dataset creation

Our theta-join workload consists of 12 manually crafted theta-join queries. None of the queries contains equality predicates (=). Hence, common join algorithms and optimizations do not apply for any of them. Instead, all predicates are based on $<, \leq, \neq, \geq, >$. The number of predicates in the queries varies between 2 and 13: TPC-H (5,2), DataSF (2,4,3), Flight (3,4,5,4), and Cloud (5,13,4). We always execute each query with two warm-up executions and report the arithmetic mean of the last five executions. A^{2}DB's theta-join algorithm, the base recordsets and our theta-join SQL queries are available online[10].

6.1 Equi-Join vs. Theta-Join

To demonstrate the remarkable performance gap between equi-join and theta-join executions, our first experiment compares the performance of an equi-join with the performance of a theta-join. To create the equi-joins for this comparison, we take the two TPC-H queries from our theta-join workload and exchange all their non-equality operators with equality operators. Table 2 shows the measured execution times on the TPC-H datatset.

Recordset	Dataset	Query	Results	PSQL$^\diamond$	SparkSQL	Redshift*	A^{2}DB
TPC-H	orig.	Q1	0	†	29,345,373	**18,630,581**	**9,371,830**
		Q1-Eq	6,366,031	**25,616**	**8,998**	332,475	158,315
		Q2	30,980,486	†	24,839,845	31,970,932	**440,435**
		Q2-Eq	833,567	**12,942**	**5,774**	4,483,270	64,198

†: Timeout after 24 hours *: 2 instead of 20 hyper-threads ◇: 1 instead of 11 nodes

Tab. 2: Query Execution time (in ms) comparison of Equi- vs. Theta-Join

The results show that the equi-joins perform orders of magnitude better than the theta-joins on all systems. This is because the systems can use sophisticated equi-join algorithms, such as (distributed) sort-merge joins, and, therefore, do not need to compare all tuple pairs. Note that the performance gain for equi-joins is not necessarily tied to smaller results: Q1's result gets larger when being turned into an equi-join, while Q2's result gets smaller, and in both cases the equi-join is faster. Even though A^{2}DB's theta-join algorithm is not optimized for

[10] `https://hpi.de/naumann/s/a2db-theta-joins` (30-November-2020)

equality predicates, it also achieves considerably faster execution times for equi-joins. For theta-join query Q2, A^2DB is clearly more efficient than all state-of-the-art competitors.

6.2 Query Performance

For a broader performance comparison of A^2DB and its competitors, we now measure the query times for all 12 theta-join queries on different subsets of our evaluation datasets. The results of this experiment are listed in Table 3.

The query times show that A^2DB significantly outperforms existing single-node DBMSs, such as PostgreSQL, and distributed batch-processing systems, such as Apache Spark, in processing highly selective theta-join queries. PostgreSQL in particular struggles to answer many theta-join queries within 24 hours that A^2DB can process in minutes. Considering the setup differences for Redshift and A^2DB, which is that Redshift has 10 times fewer threads but also over-optimizes on its hardware, both systems compete quite well. As we know that Redshift does not use join techniques, join operators or join plans optimized for theta-joins, we can conclude that its technical optimizations can actually compete with A^2DB's algorithmic optimizations. However, A^2DB clearly outperforms Redshift on some queries, such as TPC-H-Q2, Flight-Q1, and Flight-Q4, which are particularly selective. A^2DB's selectivity calculations for the individual predicates comes at the expense of extra processing time (e. g., Cloud-Q2), but the overhead is usually negligible w. r. t. the high and quadratic tuple matching costs (e. g., TPC-H-Q2).

A^2DB performs particularly well on TPC-H-Q2, Flight-Q1 and Flight-Q4, because it successfully identifies the most selective predicate, e. g., `p_retailprice >= l_extendedprice` for TPC-H-Q2 with a selectivity factor of about 1%, and prunes the candidates accordingly. On most other queries, such as SF-Q2, no single, super selective predicate exists and A^2DB needs to combine predicates for candidate pruning. With 13 predicates, query Cloud-Q2 is the largest query in the workload. Interestingly, neither PostgreSQL nor Redshift show a significant performance difference on Cloud-Q2 compared to the other queries; A^2DB is more affected by this high number of predicates, because the join matrix calculations take a larger share of the entire query processing time. In summary, A^2DB performs best on highly selective theta-join queries with possibly few predicates, which is exactly the kind of theta-join query that we observe in most use cases.

6.3 Scaling Follower Nodes

To utilize all available resources for query answering, A^2DB parallelizes and distributes the theta-join processing across a cluster of compute nodes. We now evaluate A^2DB's horizontal scalability by measuring the query execution time for both TPC-H queries on an increasing number of follower nodes to evaluate the effectiveness of the distribution. The

Dataset	Subset	Query	PostgreSQL◇	SparkSQL	Redshift*	A²DB
TPC-H	100k	TPC-H-Q1	2,392,800 ±1.2%	87,587 ±02.7%	**4,770** ±00.9%	**4,497** ±04.7%
		TPC-H-Q2	2,107,905 ±1.6%	79,581 ±32.5%	9,038 ±07.8%	**248** ±15.7%
	500k	TPC-H-Q1	56,685,284 ±0.0%	231,617 ±03.8%	**120,770** ±01.4%	**88,944** ±00.7%
		TPC-H-Q2	54,921,837 ±0.0%	402,951 ±05.7%	221,565 ±01.6%	**3,461** ±04.2%
	1M	TPC-H-Q1	†	863,338 ±02.4%	**490,846** ±00.8%	**335,696** ±00.9%
		TPC-H-Q2	†	752,133 ±04.0%	886,364 ±00.1%	**12,572** ±00.6%
	original	TPC-H-Q1	†	29,345,373 ±00.0%	**18,630,581** ±00.0%	**9,371,830** ±00.6%
		TPC-H-Q2	†	24,839,845 ±00.0%	31,970,932 ±00.0%	**440,435** ±00.8%
SFData	100k	SF-Q1	19,745 ±0.1%	10,814 ±05.6%	**381** ±03.7%	**374** ±04.5%
		SF-Q2	1,395,888 ±4.3%	107,340 ±01.5%	**6,044** ±00.3%	9,228 ±07.2%
		SF-Q3	1,374,122 ±4.3%	101,629 ±03.5%	**4,710** ±03.7%	6,861 ±03.3%
	500k	SF-Q1	509,051 ±0.1%	24,232 ±11.9%	**8,343** ±02.3%	**4,962** ±03.7%
		SF-Q2	34,557,816 ±0.7%	263,941 ±03.5%	**149,412** ±00.0%	**124,786** ±05.4%
		SF-Q3	32,563,594 ±0.0%	257,551 ±01.5%	**110,397** ±03.1%	**99,494** ±04.0%
	original	SF-Q1	1,855,169 ±0.1%	67,183 ±10.7%	**65,422** ±21.6%	**16,849** ±01.5%
		SF-Q2	†	878,780 ±02.8%	**554,790** ±01.1%	**230,320** ±65.6%
		SF-Q3	†	836,345 ±00.8%	**413,278** ±00.4%	**316,181** ±06.1%
Flight	100k	Flight-Q1	1,191,327 ±1.0%	120,986 ±57.6%	73,647 ±00.4%	**280** ±48.9%
		Flight-Q2	1,304,863 ±2.4%	132,064 ±59.9%	73,395 ±00.2%	**4,844** ±03.3%
		Flight-Q3	1,088,810 ±2.0%	152,851 ±31.4%	36,106 ±00.7%	**675** ±03.1%
		Flight-Q4	1,327,722 ±2.5%	155,010 ±35.7%	74,711 ±00.2%	**226** ±21.7%
	500k	Flight-Q1	29,650,984 ±0.4%	1,043,191 ±04.2%	126,333 ±02.8%	**1,329** ±07.0%
		Flight-Q2	16,834,932 ±0.6%	1,231,222 ±04.1%	**129,090** ±03.0%	**91,106** ±02.0%
		Flight-Q3	17,578,355 ±0.7%	301,850 ±04.7%	**72,648** ±03.3%	**11,569** ±01.8%
		Flight-Q4	16,777,141 ±0.6%	1,041,470 ±15.6%	128,522 ±02.9%	**2,000** ±05.0%
	1M	Flight-Q1	†	800,759 ±03.5%	503,496 ±00.6%	**2,318** ±05.6%
		Flight-Q2	67,258,984 ±0.0%	925,637 ±02.1%	**516,183** ±00.6%	**347,167** ±01.1%
		Flight-Q3	74,540,859 ±0.0%	870,808 ±05.6%	**295,086** ±01.2%	**50,471** ±01.1%
		Flight-Q4	66,986,894 ±0.0%	855,871 ±03.7%	514,542 ±00.7%	**8,802** ±10.3%
	original	Flight-Q1	†	38,000,013 ±02.8%	27,037,084 ±00.0%	**224,395** ±04.6%
		Flight-Q2	†	42,649,266 ±00.0%	**27,662,430** ±00.0%	**18,298,443** ±00.6%
		Flight-Q3	†	40,199,029 ±00.0%	**15,917,689** ±00.0%	**2,556,697** ±00.8%
		Flight-Q4	†	39,766,269 ±00.0%	28,007,283 ±00.0%	**433,267** ±04.5%
Cloud	100k	Cloud-Q1	1,162,777 ±0.0%	173,795 ±32.0%	65,921 ±02.3%	**2,634** ±10.9%
		Cloud-Q2	1,142,804 ±0.4%	223,127 ±27.5%	67,101 ±01.6%	**3,987** ±02.3%
		Cloud-Q3	1,178,760 ±0.0%	154,900 ±39.3%	73,966 ±24.5%	**2,824** ±03.4%
	500k	Cloud-Q1	17,881,648 ±0.8%	365,068 ±59.0%	**115,802** ±03.3%	**48,062** ±01.2%
		Cloud-Q2	20,666,685 ±0.1%	356,302 ±11.0%	**125,579** ±03.1%	**57,959** ±02.8%
		Cloud-Q3	24,579,790 ±0.3%	329,993 ±06.0%	**200,597** ±31.5%	**52,287** ±01.3%
	1M	Cloud-Q1	73,539,017 ±0.0%	940,311 ±02.3%	**460,299** ±00.8%	**177,332** ±00.2%
		Cloud-Q2	84,057,122 ±0.0%	1,115,216 ±09.2%	**497,687** ±00.5%	**224,698** ±01.0%
		Cloud-Q3	†	947,624 ±03.7%	**897,059** ±35.2%	**192,318** ±00.2%
	5M	Cloud-Q1	†	19,298,715 ±00.0%	**12,129,946** ±00.1%	**3,985,490** ±00.3%
		Cloud-Q2	†	23,178,835 ±00.0%	**11,521,068** ±00.0%	**5,519,018** ±01.1%
		Cloud-Q3	†	19,259,093 ±00.0%	**11,058,125** ±04.7%	**4,407,499** ±00.1%

†: Timeout after 24 hours *: 2 instead of 20 hyper-threads ◇: 1 instead of 11 nodes

Tab. 3: Average query execution times in ms (and maximum measurement deviations) over five measurements for different subsets of all datasets; on average, the measurement deviations are 0.8% for PostgreSQL, 11.4% for SparkSQL, 3.8% for Redshift and 4.0% for A^2DB (if we exclude measurements with sub-second duration). The best execution times are highlighted.

minimal cluster configuration has one leader and one follower node; the largest tested cluster has one leader and eleven follower nodes. We execute query *TPC-H-Q1* (empty result) on *TPC-H 500k* and query *TPC-H-Q2* (relatively large result) on *TPC-H 1M*. Figure 8 plots the query execution times in milliseconds and a reference line for *ideal*, i. e., linear scalability. The measurements for both queries show that the proposed theta-join processing scales linearly with the number of follower nodes; for this reason, we conclude that A^{2}DB's workload distribution strategy works well. Furthermore, the higher communication costs for larger cluster setups have no major impact on the overall performance, which underlines our observation that A^{2}DB's theta-join processing is CPU bound.

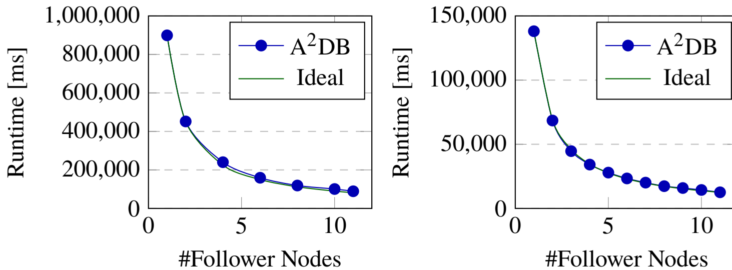


Fig. 8: *TPC-H-Q1* on *TPC-H 500k* and *TPC-H-Q2* on *TPC-H 1M* with varying cluster sizes.

6.4 Work Stealing

As followers sometimes finish their work earlier than others, A^2DB implements *work assisting* (WA) and *work stealing* (WS) to keep all nodes well utilized. The next experiment evaluates the effectiveness of the two strategies by comparing the query times with these optimizations to the ones without them. The measurements in Table 4 show that the benefit of balancing workload at the end of the join processing is 6–10% query time reduction. Hence, both strategies effectively accelerate the join processing; the lively redistribution of work between the nodes at the end succeeded to keep all nodes busy.

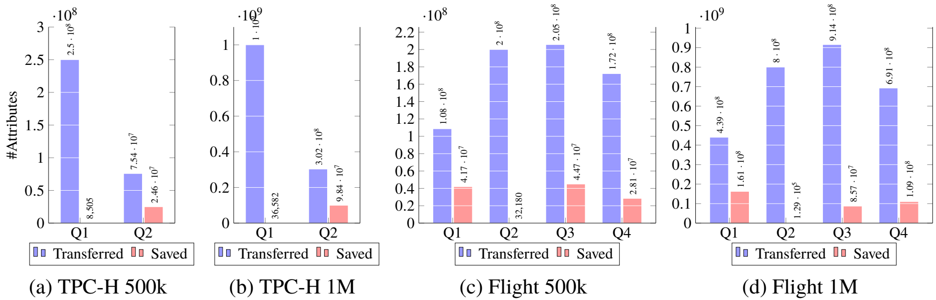
Recordset	Dataset	Query	Without WA/WS	With WA/WS	Difference
TPC-H	100k	Q1	95,030	88,944	**- 6,4%**
		Q2	3,873	3,461	**- 10,6%**
	1M	Q1	373,410	335,696	**- 10,1%**
		Q2	13,596	12,572	**- 7,5%**

Tab. 4: Query runtimes on TPC-H with and without *work assisting* and *work stealing*.

6.5 Context-Specific Attribute Provisioning

To reduce the network overhead when fetching remote data, A^{2}DB applies *context-specific attribute provisioning* to send only required values. We now evaluate the effectiveness of

this strategy by measuring the number of attribute values that are transferred and the number of non-relevant attribute values that are filtered. Figure 9 visualizes these numbers for the TPC-H and Flight queries. The measurements show that with context-specific attribute provisioning, we save about 0% (TPC-H-Q1) to 27% (Fligh-Q1) values on network traffic. Hence, the savings are dataset-specific, but can be quite substantial. Although not all queries profit from the filtering, most queries in our workload filter at least 10% values in this way.



(a) TPC-H 500k (b) TPC-H 1M (c) Flight 500k (d) Flight 1M

Fig. 9: Amount of transferred (blue) and saved (red) attributes values for the TPC-H and Flight queries with the *context-specific attribute provisioning*.

7 Summary

In this paper, we proposed a novel theta-join algorithm that accelerates the processing of selective theta-join queries via predicate-based candidate pruning and reactive workload distribution. Our experiments show that with these (and probably also further) optimizations, theta-joins can be processed orders of magnitude faster than state-of-the-art join strategies in modern data processing engines. In this way, we motivate a more careful optimization of theta-joins beyond naive nested-loop joins in modern database management systems. The selectivity-based predicate selection approach and the strategy-driven join technique can also be used to extend existing theta-join algorithms, such as IE-Join [Kh15] or 1-Bucket-Theta [Ok11], so that they can handle arbitrary many join predicates as well. Although such combinations could lead to further improvements, their construction and evaluation is not in the scope of this paper and we have to leave them to future work.

References

[AGN15] Abedjan, Z.; Golab, L.; Naumann, F.: Profiling Relational Data: A Survey. The VLDB Journal 24/4, 2015.

[Ar15] Armbrust, M.; Xin, R. S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J. K.; Meng, X.; Kaftan, T.; Franklin, M. J.; Ghodsi, A.; Zaharia, M.: Spark SQL: Relational Data Processing in Spark. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2015.

[BC81] Bernstein, P. A.; Chiu, D.-M. W.: Using Semi-Joins to Solve Relational Queries. Journal of the ACM 28/1, 1981.

[Be18] Bernstein, P. A.: Actor-Oriented Database Systems. In: Proceedings of the International Conference on Data Engineering (ICDE). 2018.

[Bo07] Bohannon, P.; Fan, W.; Geerts, F.; Jia, X.; Kementsietsidis, A.: Conditional Functional Dependencies for Data Cleaning. In: Proceedings of the International Conference on Data Engineering (ICDE). 2007.

[Co17] Cong, G.; Fan, W.; Geerts, F.; Jia, X.; Ma, S.: Improving Data Quality: Consistency and Accuracy. In: Proceedings of the VLDB Endowment. 2017.

[Co79] Codd, E. F.: Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems (TODS) 4/4, 1979.

[ESW78] Epstein, R.; Stonebraker, M.; Wong, E.: Distributed Query Processing in a Relational Data Base System. In: Proceedings of the International Conference on Management of Data (SIGMOD). 1978.

[Go75] Gotlieb, L. R.: Computing Joins of Relations. In: Proceedings of the International Conference on Management of Data (SIGMOD). 1975.

[Gu15] Gupta, A.; Agarwal, D.; Tan, D.; Kulesza, J.; Pathak, R.; Stefani, S.; Srinivasan, V.: Amazon Redshift and the Case for Simpler Data Warehouses. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2015.

[Kh15] Khayyat, Z.; Lucia, W.; Singh, M.; Ouzzani, M.; Papotti, P.; Quiané-Ruiz, J.-A.; Tang, N.; Kalnis, P.: Lightning Fast and Space Efficient Inequality Joins. In: Proceedings of the VLDB Endowment. 2015.

[KNG18] Koumarelas, I.; Naskos, A.; Gounaris, A.: Flexible partitioning for selective binary theta-joins in a massively parallel setting. Distributed and Parallel Databases 36/2, 2018.

[ME92] Mishra, P.; Eich, M. H.: Join Processing in Relational Databases. ACM Computing Surveys 24/1, 1992.

[Ok11] Okcan, A.: Processing Theta-Joins Using MapReduce. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2011.

[SSP19] Schmidl, S.; Schneider, F.; Papenbrock, T.: An Actor Database System for Akka. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW). 2019.

[VG84] Valduriez, P.; Gardarin, G.: Join and Semijoin Algorithms for a Multiprocessor Database Machine. ACM Transactions on Database Systems (TODS) 9/1, 1984.

[ZCW12] Zhang, X.; Chen, L.; Wang, M.: Efficient Multi-Way Theta-Join Processing Using MapReduce. In: Proceedings of the VLDB Endowment. 2012.