Accelerating Complex Analytics using Speculation

Panagiotis Sioulas[⋆] Viktor Sanca[⋆] Ioannis Mytilinis[⋆] Anastasia Ailamaki[⋆] [‡]
[⋆]EPFL [‡]RAW Labs SA
firstname.lastname@epfl.ch

ABSTRACT

Analytical applications, such as exploratory data analysis and decision support, process complex workloads that include sequences of inter-dependent queries. While modern OLAP systems exploit data parallelism, dependencies force execution ordering constraints that severely limit task parallelism. The serialization of tasks leads to long query response times and under-utilization of resources.

We propose a new query processing paradigm that accelerates inter-dependent queries using speculation. As when used in OLTP or in computer micro-architecture, speculative execution helps increase parallelism and improve scheduling efficiency. Nevertheless, analytics present unique challenges in making the right speculative execution decisions, in validating predictions and in repairing results. We enable fast and accurate predictions through approximate query processing (AQP), and efficiently validate speculations through a new streaming join operator. In case of mispredictions we do not discard progress, but apply corrective actions to incrementally repair the result. Our experiments over the TPC-DS benchmark show that, even though speculation adds work, it improves task parallelism, queries run faster, and more importantly, the speedup is increased as a function of query complexity.

1 INTRODUCTION

Academia and industry capitalize on data abundance to drive decision support. To efficiently extract deep insights from data, stakeholders retrieve as much information as possible in a single complex query [32]. Advanced SQL features such as views, nested queries and control flow allow users to express such complex analytical workflows monolithically. Hence, a single analytical query plan is typically translated into multiple inter-dependent sub-queries. The *dependent* parts of the plan require access to the full results of the *dependees* before they can execute. Clearly, dependencies enforce serialization and reduce task parallelism.

Limited task parallelism, in turn, hurts scalability. Modern analytics usually take place in large, distributed infrastructures, where the abundance of available resources enables data parallelism. Amdahl's law, however, implies that data parallelism by itself is insufficient, as an increase in resources often results in diminishing returns [10]. Moreover, if the dataset cannot scale out to the entire infrastructure, resources remain idle. The combination of data and task parallelism pushes further the scalability wall and enables better resource utilization. The following example query helps demonstrate the impact of dependencies:

```
Q1: SELECT COUNT(*) FROM store_sales, date_dim
WHERE ss_quantity > 90 AND
ss_sold_date_sk = d_date_sk AND
d_date BETWEEN '3/18/98' AND '6/16/98'
AND ss_sales_price > (SELECT 1.2*AVG(cs_sales_price)
FROM catalog_sales WHERE ss_item_sk=cs_item_sk)
```

The outer query computes an aggregate over the `store_sales` table. However, for filtering the tuples, we have to compute the corresponding correlated aggregate of the inner sub-query. The dependency on the inner sub-query serializes the two tasks and delays execution.

While `Q1` has a single dependency, in reality we may encounter arbitrarily complex dependencies. In exploratory analysis – a common task for analysts and computer scientists – the result of each query parameterizes the next. If the decision-making algorithm is known, rather than ad-hoc, control-flow constructs can express exploration as a deep decision tree. The final outcome of the process lies in a single leaf of the tree, and tree traversal depends on the results of a sequence of queries.

In this work, we show how to relax dependencies and compute the inner and outer sub-queries in parallel through the use of speculative execution. Speculative execution is an established technique in OLTP systems [4] and also in computer architecture [12] superscalar CPUs use branch prediction to *speculate* the next instruction to fetch and execute. After evaluating the condition, the CPU *validates* the prediction: correct predictions boost the utilization of pipeline slots and instruction-level parallelism, whereas *mispredictions* flush the pipeline and restart execution at the correct branch, incurring a delay.

Inspired by this model, we adapt speculative execution to complex analytics. As in the example of `Q1`, dependencies often occur as conditions that involve the result of the inner sub-queries; the branch prediction paradigm naturally fits the problem. Nevertheless, speculative execution is riskier and not as straightforward to apply in analytics: (i) in contrast to microprocessors, where speculation fetches a single instruction, branch prediction in analytics forwards large amounts of data to the execution path of the outer sub-query; therefore, mispredictions are much more expensive. (ii) Validation of the prediction is not supported at the hardware level but is implemented by an expensive join. (iii) Finally, to repair mispredictions, we need to re-compute a complex OLAP query, instead of simply flushing the pipeline.

At the same time, there are also some opportunities. First, predictions in analytics are data-dependent. We show that by using approximate query processing (AQP) as a branch predictor, we enable fast and educated predictions. Second, in case of mispredictions, analytics present opportunities for more fine granular actions than accept/re-compute the final result. Our mechanism triggers

11th Annual Conference on Innovative Data Systems Research (CIDR '21), January 10-13, 2021, Chaminade, USA.

some updates selectively rather than discarding the entire progress and restarting execution.

The contributions of this paper can be summarized as follows:

- We propose a framework based on speculative execution for accelerating complex OLAP queries. By relaxing dependencies, we improve task parallelism and enable data and operator sharing in cases where otherwise sharing could not be applied. Our experiments show that we gain an average speedup of 1.6× for TPC-DS queries.
- We propose the use of AQP techniques as query-aware branch predictors and show that our predictions offer both low latency and high accuracy. Speculation is a novel application for AQP that is fertile for future research.
- To offset the challenge of efficient validation, we motivate a new streaming join operator that exploits AQP and the opportunities for early validation. By selectively correcting predictions, our operator renders validation faster by an order of magnitude.

2 DEPENDENCIES IN OLAP QUERIES

First, we define the dependencies that exist in analytical workloads and can be efficiently resolved by speculative execution. In short, they should satisfy two properties: (i) be involved in a cross-query condition, and (ii) be introduced by a pipeline breaker.

Cross-query conditions. The proposed framework requires the results of inner sub-queries to occur in corresponding outer sub-queries only as part of conditional expressions. For example, in `Q1` the dependency is introduced by the condition:

```
ss_sales_price > (SELECT 1.2*AVG(cs_sales_price)..)
```

Cross-query conditions are common in nested SQL queries and appear in the WHERE, HAVING or CASE WHEN clauses. They restrict speculations to boolean, rather than continuous predictions.

Pipeline breakers. In database literature, a pipeline breaker is an operator that fully materializes at least one of its inputs [21], such as the hash or sort-based join.

The two mentioned properties define *stage breakpoints* and *stages*.

Definition 1 (Stage Breakpoint). *A materialized input of a pipeline breaker is a stage breakpoint, if it participates only in cross-query conditions.*

Stage breakpoints logically partition the query plan into *stages* that have different intermediary datasets as inputs.

Definition 2 (Stage). *A stage is a sub-tree of the query plan, where its root node is either an operator whose result is a stage breakpoint or the root of the query plan, and its leaves are either base relations or stage breakpoints.*

For example, consider the query plan of Fig. 1. The red nodes produce stage breakpoints and the plan is partitioned into four stages. The joins in S_4 inject the cross-query conditions. The following definition introduces dependencies:

Definition 3 (Dependency). *Let a stage of execution S_i that ends with stage breakpoint B and materializes a dataset D. Let also a stage S_j of the query plan with $S_j \neq S_i$. A dependency between S_i and S_j exists iff D belongs to the inputs of S_j.*

As pipeline breakers require full materialization, inter-dependent stages need to be serialized. Given sufficient resources, the total

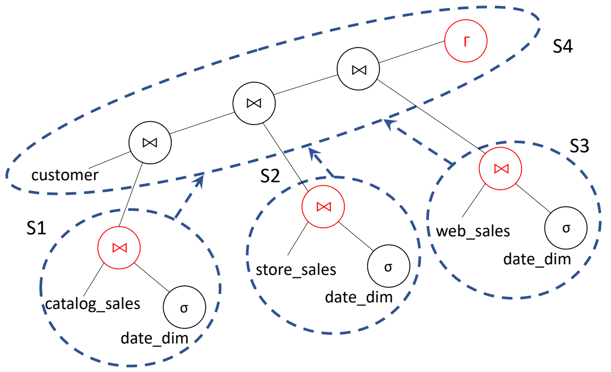


Figure 1: Example of query plan with dependencies and the corresponding stage dependency graph (blue dashed lines).

execution time for the dependency graph of Fig. 1 (blue dashed lines) is $T_1 = t_{S_4} + max\{t_{S_1}, t_{S_2}, t_{S_3}\}$, where t_{S_i} denotes the execution time of S_i. As S_4 depends on the other three stages, it needs to wait for the slowest of S_1, S_2, S_3 in order to be executed. The aim of this work is to relax dependencies and allow inter-dependent stages to either run in parallel or share data/operators. In our example, we would overlap the execution of S_4 with the other stages and ideally, this would produce a running time of $T_2 = max\{t_{S_1}, t_{S_2}, t_{S_3}, t_{S_4}\}$. Then, the maximum possible speedup $\frac{T_1}{T_2}$ would be 2× in the case when $t_{S_4} = max\{t_{S_1}, t_{S_2}, t_{S_3}\}$. By generalizing, speculative execution can claim an $O(L)$ speedup, where L is the length of the longest path in the dependency graph. Thus, the potential benefits are proportional to the depth of chained dependencies.

3 DATA VS TASK PARALLELISM

In the example of Fig. 1, we claim a speedup of up to 2× by overlapping two stages that normally would have to be serialized due to the dependency between them. While the potential gain is obvious from a theoretical viewpoint, it needs more discussion in terms of practical performance evaluation.

Data analytics systems are data-parallel, meaning that they partition data across multiple workers. These workers operate in parallel and can scale the execution of a single stage to utilize the whole cluster. Since parallelism exists even within a single stage, the question that naturally arises is why is this insufficient and how we benefit by overlapping stages and increasing task parallelism.

For many workloads, data parallelism scales sublinearly the execution and an increase in resources results in diminishing returns. This happens due to common primitives in analytical processing that introduce parallelization overheads, and hence, the corresponding stages hit a scalability wall. In this Section, we give examples of such primitives and discuss some specific cases where we can decrease the amount of resources allocated to a stage without a proportional performance decrease. Our examples are based on the in-memory execution of OLAP queries over distributed frameworks such as Spark.

Shuffling. Common database operations, such as GROUP-BY and JOIN, usually involve data exchange over the network (a.k.a shuffling) and disk accesses, when implemented on top of distributed frameworks. As the network bandwidth is usually much lower than the one of memory, it can become a bottleneck when running in-memory analytics. For the GROUP-BY case, increased parallelism may lead to increased shuffling size[1] and thus increased overhead. For high degrees of parallelism, the latency of processing a single block of data becomes too small and execution time is dominated by network/disk overheads.

While JOIN also includes shuffling, it suffers from a different kind of bottleneck; in the JOIN case, the shuffling size does not depend on parallelism. However, computation does and diminishing returns are observed due to arguments similar to Amdahl's law for parallel processing.

Broadcast join. This is typically the algorithm of preference when a *small* table is involved in a join operation. The small table is broadcasted to all partitions and a fully data-parallel probing phase follows. The broadcast of the small table incurs a synchronization overhead that limits scalability. By increasing the dedicated resources for the query, we decrease the amount of work for each executor and make probing faster. If the cost of probing becomes smaller than broadcasting the table, execution ceases to scale.

Data skew. Distributed operations require the partitioning of data and assignment to workers. Depending on the data distribution and partitioning logic, we may end up with an imbalanced situation where a few workers hold the majority of data and become *stragglers*. Increasing the amount of resources results in no benefit. We need to either change the partitioning algorithm or take advantage of the free resources to run another stage of the query.

Therefore, in cases similar to the aforementioned, executions schedules that fully utilize the cluster for a single stage are suboptimal, even when there is enough data to do so. By increasing task parallelism and overlapping stage execution, we can achieve better end-to-end execution time. Next section shows how speculative execution achieves increased parallelism even between interdependent stages.

4 SPECULATIVE EXECUTION

In this Section, we show how speculative execution relaxes dependencies, increases task parallelism, and reduces end-to-end latency. We first describe our approach for the case of a single dependency, and then generalize to complex dependency graphs.

Fig. 2 presents a high-level overview of the execution workflow. When a query is submitted, the optimizer detects the dependency between the outer and inner sub-queries and creates two separate execution paths: one for speculative execution and one for validation/repairs. The speculative execution path (solid black arrows) executes the outer query. Whenever a cross-query condition appears in the outer query, it is resolved by the branch predictor and execution continues in a pipelined fashion until results are produced. At the same time, the validation/repair execution path (red arrows) performs three steps: (i) it fully computes the inner query, (ii) it validates the speculative decisions and (iii) it repairs the results in case of mispredictions. Speculation decouples the inner and outer queries and permits the two tasks to run completely in parallel – or share data and operators if they process rows from the

[1]Assuming an early aggregation mechanism such as Hadoop Combiners is available

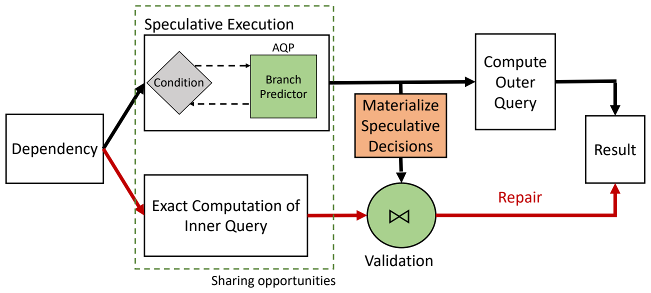


Figure 2: Overview of speculative execution for OLAP queries.

same table. In the example of `Q1`, the speculative execution path computes the COUNT over of the `store_sales` table, while the validation execution path computes the AVG of the inner query.

4.1 AQP for Branch Prediction

As in computer architecture, a branch predictor takes speculative decisions. However, the predictor's accuracy highly affects the overall efficiency of the proposed scheme. An ideal predictor would correctly evaluate the condition in zero latency. Then, as no repairs would be needed, we would completely skip computing the inner query. Also, as the predictor would respond instantly, it would introduce no delays to the speculative execution path.

Nevertheless, this is an ideal situation that cannot be realized. In practice, we have to strike a balance in the trade-off between an efficient but data-agnostic predictor, and a heavyweight but data-aware one. On the one hand, a naïve predictor, such as a random true/false evaluation, ensures low-latency decisions, but it can also yield an excessive number of mispredictions, and hence, expensive repairs. On the other hand, a more sophisticated prediction mechanism can provide accurate results and decrease the validation/repair overhead but at the cost of higher latency in the speculative execution path. Clearly, the predictor's design affects both execution paths and consequently end-to-end running time.

To hit the sweet spot in this design space, we employ approximate query processing. AQP techniques provide tunable knobs to explore the latency-accuracy trade-off and enable fast speculative decisions and low validation cost at the same time. In our running example (`Q1`), the inner query computes the AVG `cs_sales_price` of the tuples with a given `item_sk` value. An equivalent computation is to group the tuples by `item_sk` and fetch the AVG `cs_sales_price` of a specified group. Thus, the branch predictor needs to approximate the AVG for a group and check whether it satisfies the condition. GROUP-BY queries with aggregates are one of the most common applications for AQP techniques. Note also, that since the goal is to evaluate the condition and not the aggregate itself, the predictor is more robust to errors.

In this work, we motivate the use of AQP as branch predictor rather than proposing a new approximate algorithm. Moreover, unlike traditional AQP, in our setup the final query outcome is not approximate, but exact; approximations are subsequently validated and repaired. Therefore, we argue that AQP, as an in-engine accelerator, is useful even for applications that require exact results.

This opens up new research opportunities, and especially for ML-based AQP techniques [13, 17, 30] that present astonishing results in practice but have weaker theoretical guarantees.

4.2 Validating Results

For the validation, which is the most challenging part, we need to keep track of every branch prediction. For this purpose, the predictor materializes its output (orange square in Fig. 2). At the same time, the validation execution path computes the inner query and compares the results against the materialized predictions. Essentially, this comparison is a join (green circle in Fig. 2) and in case either of the two join inputs is large enough, the validation cost can become significant and offset the benefits of speculation.

For mitigating the validation cost, we propose *SVJoin*: a new join operator that works in a streaming fashion and selectively validates predictions without tampering result correctness. The streaming nature of *SVJoin* allows the two execution paths to work concurrently and modify the query result without blocking each other.

The operator takes two inputs: the exact results E from fully computing the inner query and the materialized predictions M. E takes the form of a map from the correlation attribute k (e.g., `item_sk` in `Q1`) to the corresponding true result $E(k)$ of the inner query. The materialization of M tracks information for every speculative decision. For each row i of the outer query, we maintain: (i) the correlation attribute k_i, (ii) the value v_i of the attribute that is involved in the cross-query condition (e.g., `ss_sales_price` in `Q1`), (iii) the speculative decision $d_i \in \{1, 0\}$ (i.e., whether the branch is taken or not), and (iv) all other attributes a_i^* of the row that are required for the computation of the final result (the current COUNT of `store_sales` in `Q1`). Thus, M can be formulated as a map $M : (k_i, v_i) \longrightarrow (d_i, a_i^*)$. Note that the key of M also contains v_i, since speculative decisions are taken for each distinct (k_i, v_i) pair.

SVJoin enhances symmetric join to implement $M \bowtie E$ in a streaming and selective way. As M can be too large and jeopardize performance, we show how to use the predictor's AQP semantics to reduce the size of the join. Fig. 3 illustrates our optimization for the example of `Q1`. Without loss of generality, we assume that the approximation underestimates the real AVG. We observe that in regions A and C, the speculative decision agrees with the decision based on the exact computation of the inner query; in region A both mechanisms discard the result, while in region C both of them accept it. The controversial area is B, i.e., the interval between the approximate and the real value. In the example of `Q1`, we need to validate only the decisions that involve `ss_sales_prices` in region B. The most accurate the approximation, the narrowest the interval and depending on the data distribution, the most selective the validation process can be. Therefore, *SVJoin* only needs to process the tuples from M that satisfy:

$$min\{E'(k_i), E(k_i)\} \leq v_i \leq max\{E'(k_i), E(k_i)\} \quad (1)$$

where $E'(k_i)$ is the approximate result for the given correlation key. Note that $E'(k)$ does not need to be materialized as it can be easily derived by querying the corresponding AQP synopsis. Our experiments in Section 5 indicate that the AQP-enabled selectivity can decrease validation time by an order of magnitude.

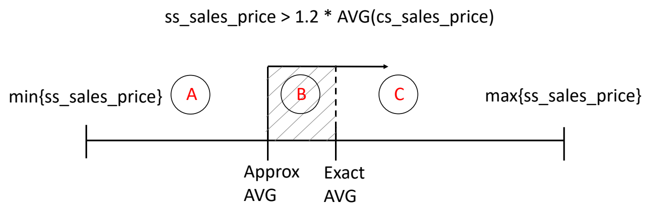


Figure 3: AQP-enabled selective validation for Q1. Only rows in region B need to be validated.

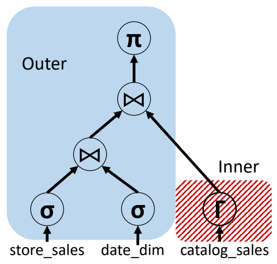
Algorithm 1: SVJoin

```
input: Tuple $t$
1  if $t$ comes from the speculative execution path then
       // $t = (k_i, v_i, d_i, a_i^*)$
2      $E(k_i)$ = probe $E$ with $k_i$;
3      if $E(k_i) \neq null$ then
4          forward $(a_i^*, \text{correct})$ in the pipeline ;
5      else
6          forward $(a_i^*, \text{speculated})$ in the pipeline ;
7          materialize $M(k_i, v_i) = (d_i, a_i^*)$
8  else
       // $t = (k_i, E(k_i))$
9      $k_i$ = correlation attribute of $t$;
10     $values_i = [min\{E'(k_i), E(k_i)\}, max\{E'(k_i), E(k_i)\}]$;
11     $S$ = probe $M$ for tuples with $k_j = k_i$ and $v_j \in values_i$;
12     if $S \neq null$ then
13         foreach $tuple \in S$ do
14             if misprediction then
15                 $\delta = (k_i, v_j, d_j, E(k_i), a_j^*)$; repair($\delta$)
16     materialize $E(k_i)$
```

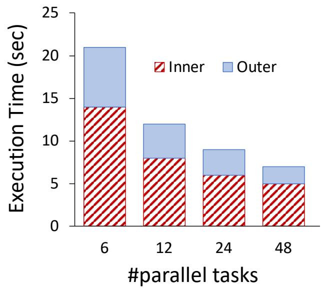
Algorithm 1 presents the outline of *SVJoin*. The algorithm works at the tuple-level and receives input both from the speculative and the validation execution paths. For tuples coming from the speculative execution path (lines 1-7), the algorithm probes E to check whether the corresponding result of the inner query has been computed. If so, it instantly validates the prediction and forwards the correct result to the next stage. Otherwise, if $E(k_i)$ has not been yet computed, to avoid breaking the pipeline, the algorithm forwards the speculation and materializes the tuple in M as explained.

For tuples coming from the validation execution path (lines 8-16), the algorithm applies selective validation; it locates which tuples are eligible for validation (line 10) and probes M for candidate matches. Of course, for efficient filtering, M should support range queries on the key (k_i, v_i). However, as range queries are supported by many systems, we do not further elaborate this point.

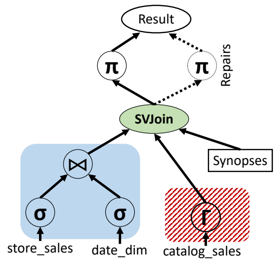
In line 15 of the algorithm, when a misprediction is detected, a δ-object, with the correct update, is constructed and forwarded to the repair mechanism. The repair mechanism identifies the results that come from wrong speculations and corrects them. To apply repairs, we define a new algebra $\mathcal{A}$. For each relational operator, the corresponding operator in $\mathcal{A}$ works over δ-objects and updates the result. The algebra infers a repair plan for each stage of the original query plan to propagate the updates to the final result.



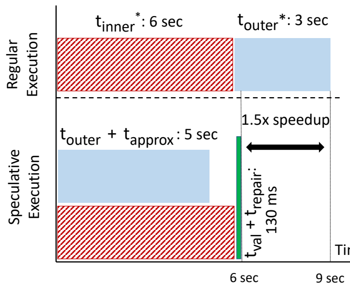
(a) Decorrelated execution plan for Q1



(b) Breakdown of regular execution over Spark



(c) Plan for speculative execution



(d) Regular vs speculative execution (24 threads per stage)

Figure 4: Execution plans and time breakdown for the running example Q1.

To sum up, *SVJoin* never blocks and for each input tuple, it has three likely outcomes: (i) correcting the speculation on-the-fly and forwarding the exact value (line 4), (ii) forwarding the speculated value (line 6), or (iii) forwarding a set of δ-objects (line 15).

4.3 Example

Our framework permits increased task-parallelism and data/work-sharing between the inner and outer query, but also incurs additional computation for approximating the inner query, validating the speculations and repairing the results. Thus, a speculation-aware optimizer should evaluate the corresponding costs and apply speculative execution only when the expected benefit outweights the overhead. Quantitatively, this happens when:

$$max\{t_{inner}, t_{approx} + t_{outer}\} + (t_{valid} + t_{repair}) < t^*_{inner} + t^*_{outer}$$

where the left-hand-side of the inequality represents the execution time of our approach and the right-hand-side the purely data-parallel execution, where different sub-queries are serialized. In this Section, we provide an execution time analysis for our running example Q1, through which we explain how the framework achieves speedup.

Fig. 4(a) depicts the plan we use for processing Q1: it comprises a GROUP-BY over the `catalog_sales` table (inner query) followed by broadcast joins with `store_sales` and `date_dim` (outer query). Fig. 4(b) shows the time breakdown for executing this plan with in-memory data over a Spark cluster of 4 machines. Details on our experimental setup are provided in Section 5. We observe that data parallelism offers diminishing returns and using more than 24 parallel tasks does not significantly reduce execution time. The cause of the scalability bottleneck is the shuffling due to the GROUP-BY of the inner query and the broadcast joins of the outer query.

In this case, our framework can provide speedup while using the same amount of resources. A speculation-aware optimizer would detect the scalability problem of each stage and instead of serializing them and investing 48 cores to each sub-query, it would assign only 24 cores to each sub-query and execute both in parallel. The corresponding speculative plan is illustrated in Fig. 4(c). Fig. 4(d) presents the timeline of the speculative versus the regular data-parallel execution. For the purely data-parallel execution, the end-to-end execution time is $t^*_{inner} + t^*_{outer} = 9sec$. During speculative execution, the inner and outer sub-queries

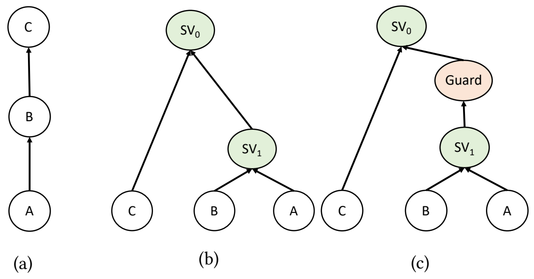


Figure 5: Speculative execution plan for a dependency path of length greater than one: (a) dependency graph, (b) plan with *SVJoins*, (c) complete plan with *SVJoins* and *Guards*

run completely in parallel and a validation-repairs phase follows ($max\{t_{inner}, t_{approx} + t_{outer}\} + (t_{valid} + t_{repair}) = 6sec$). Notice that the execution time of the speculative outer sub-query is increased compared to the purely data-parallel execution, as it also includes the computation of approximations(t_{approx}) and the materialization of the speculative decisions. Moreover, we observe that validation is an order of magnitude faster than the actual execution of the two sub-queries. Validation has very low execution time as it evaluates decisions only for the join's probe input and only for the range of `ss_sales_price` values that satisfy Inequality (1). Overall, the speedup we achieve for Q1 is 1.5× compared to the purely data-parallel execution.

4.4 Complex Dependency Graphs

So far, we have discussed the case where a single dependency exists in the plan. However, the dependency graph can be an arbitrarily complex DAG of stages. The generalization demands to extend the above description for the following cases: (i) a single outer query depends on more than one inner queries, and (ii) the dependency graph contains paths of length greater than one.

For the first case, where a stage depends on multiple inner queries (e.g., Fig. 1), we have extended *SVJoin* to receive multiple inputs: one for each inner query's validation path, and one for the speculative path that performs branch predictions for all cross-query dependencies. Thus, in the general, *SVJoin* is an n-ary operator.

Speculative execution over deep dependency graphs is defined in a recursive way in Algorithm 2. Let us explain the idea through an example. In the dependency path of Fig. 5(a), We start from the output stage C and insert an *SVJoin* SV_0 in order to resolve the outermost dependency. SV_0 has two inputs: (i) the materialized speculative decisions for stage C and (ii) the true results for the query part over $A \rightarrow B$. As the second input is not a base relation, we add another *SVJoin* SV_1 that resolves the $A \rightarrow B$ dependency. The resulting speculative plan is depicted in Fig. 5(b). However, deep streaming pipelines can harm repair stability and overall performance predictability. As *SVJoin* is a streaming operator, it can forward speculative results from stage B to the validation execution path of SV_0 before SV_1 validates them. The result is multiple unnecessary and overlapping repairs.

To remedy this, we add a special *Guard* operator that acts as a "glue" between successive *SVJoins* and permits only correct tuples to flow between them. The type of an *SVJoins's* output tuples defines the *Guard's* behavior. If a tuple contains a correct result, *Guard* directly forwards it to the next *SVJoin*. If a tuple is a δ-object then *Guard* applies the $\mathcal{A}$ algebra, corrects the result and forwards it. Finally, if the tuple is a speculation, *Guard* materializes it and waits until it is corrected before emitting it to the next *SVJoin*. Fig. 5(c) illustrates the speculative plan with the *Guard* included.

Algorithm 2: SVJoinConvert

```
input: Dependency graph $G$, Graph Node $s$
// $s$ is the output stage of $G$
1 $SV$ = new SVJoin(); $SV$.inputs().add($s$) ;
2 foreach parent $p$ of $s$ do
3     if $p$ has no parents then
4         $SV$.inputs().add($p$) ;
5     else
6         $g$ = new Guard(); $SV_2$ = new SVJoin() ;
7         $g$.inputs().add($SV_2$); $SV$.inputs().add($g$) ;
8         SVJoinConvert($G, p$) ;
```

4.5 Imperative Query Processing

The proposed framework accelerates not only nested queries but also imperative programs that combine multiple relational queries using imperative constructs, such as variables, conditional statements and loops. Imperative programs can express complex analytical workloads, including exploratory data analysis and iterative algorithms such as k-Means [31].

DBMS support imperative programs that may contain SQL statements through stored procedures and user-defined functions. More recently, to satisfy the demand for interweaving relational processing with diverse operations offered by linear algebra, machine learning, and visualization frameworks, data management has become increasingly integrated to programming languages. The APIs of library-based frameworks (e.g., Pandas, dplyr) or embedded databases (e.g., SQLite, DuckDB [25]) inject analytical queries into programs written in the host language. In all cases, queries handle data-intensive processing, whereas imperative constructs orchestrate data and control flow.

Data and control flow create dependencies between queries and serialize execution. On the one hand, similar to the case of nested sub-queries, data flow dependencies occur when a query needs to access the results of another query. Imperative programs achieve data flow using variables that temporarily store query results; then, subsequent queries use results assigned to variables. The data flow dependency serializes the queries involved. On the other hand, control flow dependencies occur in conditional statements and loops, when the branch condition contains the results of one or more queries. Depending on the branch decision, the imperative program executes different queries(e.g., exploratory data analysis). Hence, it waits for the queries involved in the branch decision to finish, before it starts executing any further queries. Both types of dependencies co-exist in imperative programs.

Imperative programs contain a large number of queries, but permit limited task-parallelism. By relaxing data and control flow dependencies, speculative execution can increase task-parallelism and reduce the total execution time. Using temporary tables is equivalent to nested queries that our framework accelerates; in fact, optimizations over imperative programs often produce nested queries by inlining relational expressions [7, 26]. We further extend the framework to handle control flow dependencies. By predicting branch decisions using approximate query results, the framework eagerly runs queries that are likely to execute. To handle mispredictions, it restarts execution from the first wrongly predicted branch; then, it benefits from all preceding correct predictions. The framework first handles control flow dependencies, thus deciding which queries to execute, and then data flow dependencies.

4.6 Discussion

Implementation-wise, the proposed framework requires changes to both the optimizer and the execution engine. During planning, the optimizer detects dependencies and creates the stage dependency graph. Then, it runs Algorithm 2 and inserts *SVJoin* and *Guard* operators into the plan. The resulting plan comprises *SVJoins*, *Guards* and stage nodes. To exploit both pipelining within each stage and sharing across stages, we envision a just-in-time compiled execution engine [21] that performs intra- and inter-stage operator fusion.

We also consider future work along three axes: i) approximations: speculative execution motivates a need for empirically accurate approximations for a wide range of queries. Moreover, as there is no silver-bullet in AQP, we aim to work on an optimizer that selects the most suitable AQP technique for the query at hand. ii) modern hardware: the proposed framework combines multiple processing paradigms (e.g., approximations, exact processing, incremental updates, etc.) with different performance characteristics. We plan to use hardware accelerators to offload execution steps judiciously. iii) applications: we plan to use speculations for a wider class of workloads. In HTAP, for example, analytical queries ingest data from a transactional engine. To maximize freshness, the analytical engine can speculatively compute queries over stale data and materialized views, in parallel with update propagation and incremental view maintenance, and repair the result based on the update deltas.

5 EXPERIMENTS

In this Section, we validate the proposed framework with a series of experiments using the popular TPC-DS benchmark, a decision support benchmark that models a wide range of applications, including ad-hoc, reporting, iterative, and data-mining queries [19]. Such applications often process complex queries, which our framework targets. We demonstrate: (i) the end-to-end running time improvement, (ii) the importance of task parallelism in overcoming the scalability wall, and (iii) the correlation between AQP and efficient validation. By parallelizing stages and applying repairs efficiently, we increase scalability and significantly reduce the latency of query processing.

While the proposed framework can be implemented on top of any analytical engine, we chose to built the initial prototype using Spark, as it is a mature and widely used system for large-scale analytics. In general, two basic requirements should be satisfied to enable speculative execution; the underlying system should support: (i) concurrent query processing and (ii) work-sharing. Spark achieves concurrent jobs through Scala's concurrency primitives (*Futures* [1]). As Spark does not natively support work-sharing, currently we have implemented custom solutions per query.

Then, on top of Spark, we need to implement: (i) the approximate query processing algorithms for the speculations, (ii) the *SVJoin* and *Guard* operators and finally, (iii) a speculation-aware optimizer. However, in the current work the plans are manually implemented.

We envision speculative execution to be used in large highly-parallel clusters with several nodes. We expect large clusters to have sufficient main memory for storing the data. Furthermore, for analytical workloads, especially iterative algorithms and exploratory data analysis which our framework aims to accelerate, main-memory analytics achieve drastically better performance compared to disk-based analytics such as MapReduce [29]. Therefore, as our target application is main-memory analytics, in the following experiments, we store data completely in-memory.

We run experiments in a 4-node homogeneous YARN cluster. Each machine is equipped with two Intel Xeon CPU E5-2660 CPUs (2.20GHz, 8 cores), 128 GB of DRAM and SATA 3Gbps HDD configured in RAID0. A 10Gb Ethernet switch connects the nodes. The cluster configuration designates three nodes as both Spark workers and HDFS DataNodes and one node as the driver. We use Spark 2.4.6, Hadoop 2.10, JDK 1.8 and Scala 2.11.

5.1 Execution time for TPC-DS

We evaluate our framework over the TPC-DS benchmark with scale factor SF=100. For this purpose, we have implemented speculative execution for all the TPC-DS queries with nested sub-queries and cross-query conditions (9% of the benchmark). Those queries are representative of the non-trivial case of decorrelation, which resolves dependencies by transforming inner queries to GROUP-BY aggregations and by using a semi-join on the correlated attribute or aggregated value. As aggregated values only participate in the semi-join conditions, they satisfy the conditions presented in Section 2. Efficient approximation methods exist for speculation over the aggregate values. Sideways predicate passing [9] is also used wherever possible. Sub-queries cover four broad categories: i) aggregation with a cross-query filtering condition over a view (Q_1, Q_{30}, Q_{81}), ii) aggregation with a cross-query filtering condition and a JOIN on a filtered fact table (Q_{32}, Q_{92}), iii) existence with cross-query JOIN on a fact table (Q_{16}), iv) existence with cross-query JOIN on a fact table with a filtering condition (Q_{10}, Q_{35}, Q_{69}).

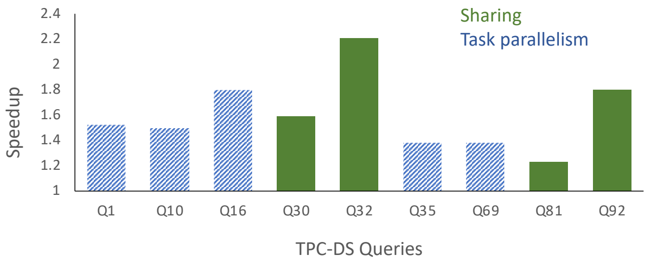


Figure 6: Query execution speedup for TPC-DS queries

Fig. 6 shows end-to-end execution time speedup when compared to the regular Spark execution. The speedup we gain ranges from 1.2× to 2.2× (1.6× on average). While we need to do extra work for validation and repairs, increased task parallelism accelerates all implemented queries. The determining factor of expected benefits for a specific query is the relative difference between the cost of the outer and inner sub-queries. If the inner and outer queries have comparable latency, parallelization provides significant speedup. By contrast, imbalance limits the opportunity for acceleration. A cost-based optimizer can identify the relative costs of the sub-queries, and predict the benefit from speculative execution.

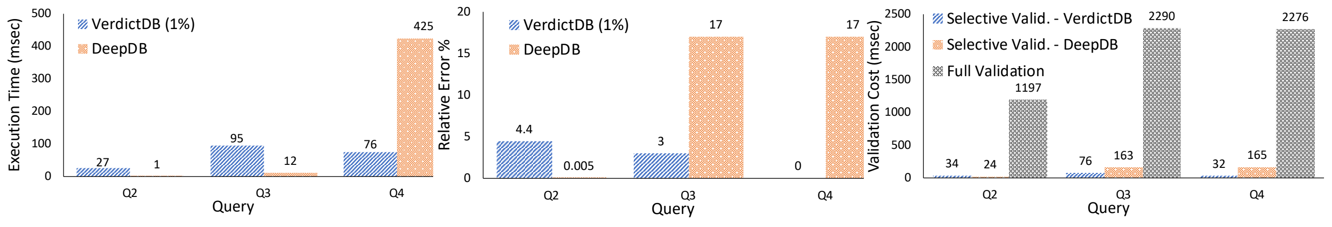
The queries with green bars in Fig. 6 are accelerated by exploiting sharing opportunities, while the rest exploit only task parallelism. In the case of sharing, our framework permits the computation of the inner query and of the speculative phase with a single pass over the data and this is where speedup comes from. Queries with blue bars overlap the execution of the inner and outer sub-queries, and by increasing task parallelism, they overcome scalability problems caused by shuffling and broadcasts and achieve reduced end-to-end time compared to the regular, data-parallel-only Spark execution.

5.2 AQP Impact

We investigate the suitability of state-of-the-art AQP techniques for branch-prediction and the correlation between AQP and the validation cost. We use offline AQP methods as they provide the highest speedup by constructing data synopses ahead of time, and evaluate three different scenarios based on the query-driven change in data distribution. Our microbenchmark considers the following inner queries:

```
Q2. SELECT 1.2*AVG(ss_list_price) FROM store_sales
Q3. SELECT ss_store_sk k, 1.2*AVG(ss_list_price)
    FROM store_sales GROUP BY ss_store_sk
Q4. SELECT ss_store_sk k, 1.2*AVG(ss_list_price)
    FROM store_sales WHERE ss_list_price>100
    GROUP BY ss_store_sk
```

Q_2 represents the base case where the aggregate value is approximated over the full column of the `store_sales` table. Q_3 performs a GROUP-BY-AVG aggregation, hence it evaluates the resilience of the approximation method to compute the data distribution based on individual grouping keys. Q_4 adds a filter value in the WHERE



(a) Latency (b) Relative Error (c) Validation Cost

Figure 7: Comparison of VerdictDB and DeepDB with respect to latency, accuracy and validation cost.

clause of the $Q3$ and further evaluates the ability of the approximation to model the subsets of data and changes in the initial data distribution. For approximations, we use VerdictDB [23] configured with 1% sampling rate and DeepDB [13].

The error-latency results are presented in Fig. 7(a)-(b). Both AQP techniques are at least an order of magnitude faster than the exact computation of the inner query, enabling fast branch predictions with sub-second responses and a speedup ranging from 24× to 159×. VerdictDB evaluates the queries over a stratified data sample. Processing less data reduces the overall execution time, while carefully constructing a stratified sample that includes the grouping columns as keys maintains the approximation quality over the queries. DeepDB constructs a Sum-Product Network based synopsis [20] which is more compact than the sample and enables faster evaluation of aggregate queries. However, in presence of GROUP-BY queries with small groups the approximation error is lower for the examined query and data size, while adding a highly selective predicate reduces the accuracy and increases the execution time. A trade-off exists between approximation latency and error: a faster approximation permits speculative execution to start earlier, while an accurate approximation reduces the cost of validation and repair. In Fig. 7(c) we quantify the impact of AQP on the validation cost. As discussed in Section 4.2, AQP enables the optimization of selective validation; the more accurate the synopsis, the more efficient the validation. We compare the validation time when each of the two AQP systems is used versus processing the full join because there is no selective access path. Both AQP systems enable highly selective joins and decrease validation time by an order of magnitude. As VerdictDB is more accurate than DeepDB when running the queries in our setup, VerdictDB's validation costs are lower. Estimating the trade-off between approximate answer latency and validation cost remains an open research milestone towards an optimizer that automatically selects the best available data summary for the speculation.

6 RELATED WORK

Nested SQL Queries. Nested queries have attracted a lot of interest, as they are a convenient way to express complex dependencies. The seminal work of Won [15] proposes rewriting queries to move the correlation predicate to the outer block. Magic decorrelation [28] is a more general technique that materializes inner query results for all outer references and then performs a join. SQL window functions have also been used to eliminate the inner subquery [32]. While window functions are efficient to implement [16], they can be applied under certain limited conditions. In [27], performance improvements are demonstrated by caching the portion of the inner query that is invariant with respect to the changing outer values. Existing database systems [3, 8] implement one or more of the aforementioned techniques. While existing techniques can completely decorrelate and flatten some queries, in the general case they eliminate redundant data accesses, but retain dependencies which prevent task-parallel execution. Speculative execution is complementary to previous research. We assume that it is applied on an already decorrelated plan that efficiently computes the inner query, thus yielding an incremental benefit.

Speculative Execution. There are two types of speculation: predictive and eager. Predictive speculation, in which this work belongs, predicts the correct path of execution and applies repairs as needed. This is the case for avoiding pipeline stalls [12] in microprocessors and for speculative concurrency control [4] in OLTP. By contrast, eager speculation executes both branches of the condition or spawns multiple identical tasks to avoid stragglers. This is inherently supported by some programming languages (e.g., Scala [1]), and by several big data frameworks (e.g., Apache Spark) [5].

AQP comprises a rich body of work in both algorithms and systems. There are many techniques such as samples [23], histograms [24], wavelets [18], sketches [6], and more recently, ML-models [13, 17, 30]. As there is no silver bullet, depending on the query type, each technique presents different accuracy-latency trade-offs. Based on whether the synopses are pre-computed or constructed as by-product of execution, we classify them as offline [2], online [14] or hybrid [22]. This work does not propose a new AQP technique and does not stand for a specific existing one. However, in order to minimize prediction latency during speculative execution, we opt for offline techniques. Similarly to selecting and using index structures, the materialization of the right synopses is part of the database tuning.

Online aggregation [11] provides approximate, progressively-refined query results by propagating tuples through operators in a controlled order. Similarly, speculative execution progressively repairs the early approximated results in the speculative path. Progressive execution is only a side-effect of the speculation, where the main effect is increasing task parallelism and exposing work-sharing opportunities. Unlike online aggregation, speculative execution focuses on reducing end-to-end latency rather than on providing early results.

7 CONCLUSION

In this paper, we propose the use of speculative execution for relaxing dependencies and parallelizing complex OLAP workloads. Our framework restores task parallelism across inter-dependent queries, by resolving dependencies through predictions. We demonstrate that AQP techniques enable the construction of accurate low-latency branch predictors. Also, we show that the cost of validating predictions is crucial and motivate a selective streaming join operator, *SVJoin*, that can accelerate validation by more than an order of magnitude. Despite introducing additional work, the framework accelerates complex TPC-DS queries by 1.6× on average. Our approach sets the foundations for a new paradigm of data-driven query processing and presents a novel application for AQP.

8 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their thoughtful comments and constructive criticism. Also, we would like to thank our friends and colleagues Stella Giannakopoulou, Haoqiong Bian, Srinivas Karthik and Bikash Chandra for their valuable feedback. This work was partially funded by EU H2020 project SmartDataLake (825041) and SNSF project "Efficient Real-time Analytics on General-Purpose GPUs", subside no 200021_178894/1. Panagiotis Sioulas is funded by the 2020 Facebook Fellowship Program.

REFERENCES

[1] Scala doc. https://www.scala-lang.org/files/archive/api/2.12.3/scala/concurrent/Future.html.

[2] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., and Stoica, I. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), pp. 29–42.

[3] Bellamkonda, S., Ahmed, R., Witkowski, A., Amor, A., Zait, M., and Lin, C.-C. Enhanced subquery optimizations in oracle. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1366–1377.

[4] Bestavros, A. *Speculative concurrency control.* Boston University, Graduate School of Arts and Sciences, Computer Science ..., 1993.

[5] Chen, Q., Liu, C., and Xiao, Z. Improving mapreduce performance using smart speculative execution strategy. *IEEE Transactions on Computers 63*, 4 (2013), 954–967.

[6] Cormode, G., and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[7] Duta, C., Hirn, D., and Grust, T. Compiling pl/sql away, 2020.

[8] Elhemali, M., Galindo-Legaria, C. A., Grabs, T., and Joshi, M. M. Execution strategies for sql subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), pp. 993–1004.

[9] Ganski, R. A., and Wong, H. K. T. Optimization of nested sql queries revisited. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1987), SIGMOD '87, Association for Computing Machinery, p. 23–33.

[10] Gray, J., and Shenoy, P. J. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000* (2000), IEEE Computer Society, pp. 3–10.

[11] Hellerstein, J. M., Haas, P. J., and Wang, H. J. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* (1997), pp. 171–182.

[12] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[13] Hilprecht, B., Schmidt, A., Kulessa, M., Molina, A., Kersting, K., and Binnig, C. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow. 13*, 7 (2020), 992–1005.

[14] Kandula, S., Shanbhag, A., Vitorovic, A., Olma, M., Grandl, R., Chaudhuri, S., and Ding, B. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data* (2016), pp. 631–646.

[15] Kim, W. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS) 7*, 3 (1982), 443–469.

[16] Leis, V., Kundhikanjana, K., Kemper, A., and Neumann, T. Efficient processing of window functions in analytical sql queries. *Proceedings of the VLDB Endowment 8*, 10 (2015), 1058–1069.

[17] Ma, Q., and Triantafillou, P. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 1553–1570.

[18] Mytilinis, I., Tsoumakos, D., and Koziris, N. Scaling the construction of wavelet synopses for maximum error metrics. *IEEE Transactions on Knowledge and Data Engineering 31*, 9 (2018), 1794–1808.

[19] Nambiar, R. O., and Poess, M. The making of tpc-ds. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, p. 1049–1058.

[20] Nath, A., and Domingos, P. M. Learning relational sum-product networks. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA* (2015), B. Bonet and S. Koenig, Eds., AAAI Press, pp. 2878–2886.

[21] Neumann, T. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment 4*, 9 (2011), 539–550.

[22] Olma, M., Papapetrou, O., Appuswamy, R., and Ailamaki, A. Taster: self-tuning, elastic and online approximate query processing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), IEEE, pp. 482–493.

[23] Park, Y., Mozafari, B., Sorenson, J., and Wang, J. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1461–1476.

[24] Poosala, V., Haas, P. J., Ioannidis, Y. E., and Shekita, E. J. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record 25*, 2 (1996), 294–305.

[25] Raasveldt, M., and Mühleisen, H. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1981–1984.

[26] Ramachandra, K., Park, K., Emani, K. V., Halverson, A., Galindo-Legaria, C., and Cunningham, C. Froid: Optimization of imperative programs in a relational database. *Proc. VLDB Endow. 11*, 4 (Dec. 2017), 432–444.

[27] Rao, J., and Ross, K. A. Reusing invariants: A new strategy for correlated queries. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (1998), pp. 37–48.

[28] Seshadri, P., Pirahesh, H., and Leung, T. C. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering* (1996), IEEE, pp. 450–458.

[29] Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Özcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow. 8*, 13 (Sept. 2015), 2110–2121.

[30] Thirumuruganathan, S., Hasan, S., Koudas, N., and Das, G. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), IEEE, pp. 1309–1320.

[31] Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Yu, P. S., Zhou, Z.-H., Steinbach, M., Hand, D. J., and Steinberg, D. Top 10 algorithms in data mining. *Knowl. Inf. Syst. 14*, 1 (Dec. 2007), 1–37.

[32] Zuzarte, C., Pirahesh, H., Ma, W., Cheng, Q., Liu, L., and Wong, K. Winmagic: Subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 652–656.