PLAQUE: Automated Predicate Learning at Query Time

YIMING LIN[*], University of California, Berkeley, USA
SHARAD MEHROTRA, University of California, Irvine, USA

Predicate pushing down is a key optimization used to speed up query processing. Much of the existing practice is restricted to pushing predicates explicitly listed in the query. In this paper, we consider the challenge of *learning* predicates during query execution which are then exploited to accelerate execution. Prior related approaches with a similar goal are restricted (e.g., learn only from only join columns or from specific data statistics). We significantly expand the realm of predicates that can be learned from different query operators (aggregations, joins, grouping, etc.) and develop a system, entitled PLAQUE, that learns such predicates during query execution. Comprehensive evaluations on both synthetic and real datasets demonstrate that the learned predicate approach adopted by PLAQUE can significantly accelerate query execution by up to 33x, and this improvement increases to up to 100x when User-Defined Functions (UDFs) are utilized in queries.

CCS Concepts: • **Information systems** → **Query operators**.

Additional Key Words and Phrases: Data Management, Query Processing

ACM Reference Format:
Yiming Lin and Sharad Mehrotra. 2024. PLAQUE: Automated Predicate Learning at Query Time. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 46 (February 2024), 25 pages. https://doi.org/10.1145/3639301

1 INTRODUCTION

Predicate pushdown based on selectivity and cost estimates is a key strategy used to optimize queries in relational databases. Pushing predicates down in a query tree could lead to significant savings by reducing the size of data that migrates to downstream operators. In this paper, we seek a new approach to query processing, entitled PLAQUE, automated *P*redicate *L*e*A*rning at *QU*ery tim*E*, that learns selective predicates *during* query execution (beyond those listed explicitly) in order to filter out tuples that would not result in any query results as early as possible during query processing. To illustrate the key idea behind PLAQUE, we examine a slightly modified and simplified version of TPC-H Query Q-10 that includes a theta-join condition. [1] In this query, the predicates o_orderdate < '1993-01-01' and p_brand = ':10' can be pushed down to orders and parts tables. However, the query contains no predicates on the lineitem table that could prune non-matching lineitem records that do not result in any query results. Thus, any query plan without a built index will scan over all records in the lineitem table.

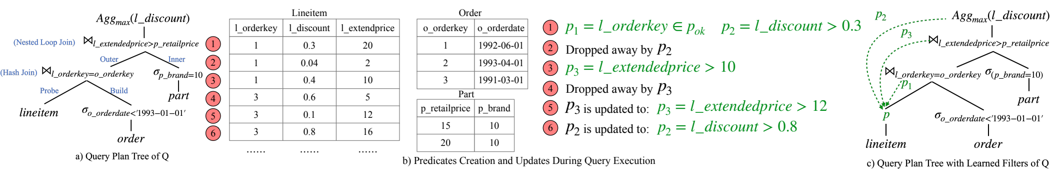
```
SELECT MAX(l_discount)
FROM part, lineitem, orders
WHERE p_retailprice < l_extendedprice AND o_orderkey = l_orderkey AND o_orderdate < '1993-
01-01' AND p_brand = ':10'
```

[*]Work done at UC Irvine.
[1]The similar query is used in previous works [23, 25] to evaluate theta-join in TPC-H benchmark.

Authors' addresses: Yiming Lin, University of California, Berkeley, USA, yiminglin@berkeley.edu; Sharad Mehrotra, University of California, Irvine, USA, sharad@ics.uci.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).
ACM 2836-6573/2024/2-ART46
https://doi.org/10.1145/3639301


$Agg_{max}(l\_discount)$
(Nested Loop Join) $\bowtie_{l\_extendedprice>p\_retailprice}$
Outer
Inner
(Hash Join) $\bowtie_{l\_orderkey=o\_orderkey}$
$\sigma_{p\_brand=10}$
Probe
Build
part
lineitem
$\sigma_{o\_orderdate<'1993-01-01'}$
order
a) Query Plan Tree of Q
1 2 3 4 5 6
b) Predicates Creation and Updates During Query Execution
p_2
p_3
p_1
p
$\sigma_{(p\_brand=10)}$
c) Query Plan Tree with Learned Filters of Q

Lineitem

l_orderkey	l_discount	l_extendprice
1	0.3	20
1	0.04	2
1	0.4	10
3	0.6	5
3	0.1	12
3	0.8	16
......	......	......

Order

o_orderkey	o_orderdate
1	1992-06-01
2	1993-04-01
3	1991-03-01

Part

p_retailprice	p_brand
15	10
20	10

1 $p_1 = l\_orderkey \in p_{ok}$ $p_2 = l\_discount > 0.3$
2 Dropped away by p_2
3 $p_3 = l\_extendedprice > 10$
4 Dropped away by p_3
5 p_3 is updated to: $p_3 = l\_extendedprice > 12$
6 p_2 is updated to: $p_2 = l\_discount > 0.8$

Fig. 1. Learned Predicates during Query Execution and the Updated Query Plan Tree of the Simplified TPC-H Q10.

Consider that all records in the lineitem table that result in an answer satisfy a predicate $l\_discount > 0.7$ - we will momentarily see how PLAQUE learns such predicates. Query execution can be significantly accelerated by pushing such predicates down in the query to filter records in the lineitem table. Only a small fraction of lineitem records will need to join with the orders and parts tables resulting in significant savings.

Several prior works have explored learning predicates, other than those specified explicitly in queries, to reduce downstream query processing. Such approaches, typically learn predicates prior to the execution of the query based, especially, on exploiting query predicates on join columns (e.g., [9, 12, 20, 22]). For instance, if the query above contained a predicate l_orderkey < 5) in addition to the other predicates listed in the query, techniques such as [26] could infer a new predicate o_orderkey < 5) which could then be used to filter tuples from the orders table to speed up the query execution. Such prior work on learning predicates, however, is of limited applicability since queries containing equi joins seldom contain additional selection predicates on the join column. This can be observed by examining such equi-join queries over several real datasets and synthetic benchmarks such as TPC-H [7] or TPC-DS [6] in which none of the equi-join queries contain additional predicates on the join columns. As such, above mentioned techniques rarely result in significant execution cost reduction of benchmark queries. An alternate strategy that empowers predicates learned ahead of query execution has been explored in [16]. In this strategy, the system maintains data statistics (e.g., min and max of columns) at the data block level which is used for sideways information passing over equi-joins accelerating query execution, especially in big-data systems such as Hive or Pig where data is partitioned across clusters. The work, however, is limited to equi-joins.

In contrast to learning predicates before query execution, some prior work [15, 21, 27] have also considered alternate strategies that, similar to PLAQUE, infer predicates to add to queries on the fly during query execution. Much of this work, however, has been in the context of hash-joins in main-memory database settings. Such strategies build summarization data structures, such as bloom filters, for the build table and use them to skip tuples in the probe table. We note that approaches that learn predicates prior to execution [9, 12, 20, 22], and those that learn predicates during execution, can be considered as complementary - they can be used in conjunction.

In this paper, we propose PLAQUE that similar to [15, 21, 27] learns predicates to add to the query during query processing. In contrast to them, PLAQUE takes a much more comprehensive, as well as, an adaptive approach to learning and using predicates in query execution. PLAQUE infers new predicates not just during execution of hash-join (as in [15, 21, 27]) but based on *a range of relational operators* including aggregation operators such as min and max, theta-joins, equi-joins, group-by operators and having conditions in queries. In PLAQUE, as query execution proceeds and records pass through operators in the query tree, the system learns new predicates to reduce downstream data processing. Such predicates learned are further refined as query processing proceeds (and more data is seen) resulting in improved filters. Predicate learning in PLAQUE occurs not just when the system uses a hash-based operator implementation (e.g., as in hash-joins) but also when nested-loop or sort-merge algorithms are used (as will be clear in Section 3). In PLAQUE, predicate learning and maintenance including predicate refinement is performed efficiently and remains a negligible part of query execution cost. PLAQUE in addition to saving computation cost by pruning unnecessary records, also supports checking of newly-learned predicates using an index-based implementation to reduce I/O costs and decides on the optimal placement of the learned filters (as we will show later, placement of the operator depends upon when it is learned and a simple rule such as pushing the predicate as far down the tree as possible may not be optimal).

Comprehensive evaluations on two benchmarks (TPCH [7] and SmartBench [13]) and one real dataset (IMDB) [2] in Section 6 demonstrate that adding the learned predicates using PLAQUE can achieve significant improvement ranging from 2x-33x, especially in queries containing expensive User-Defined-Functions (UDFs) where the improvement can be up to 100x in SmartBench [13].

Specifically, this paper makes the following contributions.

- A set of novel approaches to infer predicates during query execution from aggregate, equi join, theta join, and having conditions in the given query, and place them wisely in the given query tree to maximize the benefits from predicates pushdown using a partial-order based graphical approach.
- A system entitled PLAQUE to exploit the learned predicates using index and in-memory predicates to effectively save both I/O cost and memory footprint.
- A set of comprehensive experiments on both real and synthetic benchmarks to evaluate the effectiveness of our learned predicates. We further test the learned predicates on queries with UDFs to demonstrate their broader applicability.

2 PLAQUE OVERVIEW

PLAQUE learns predicates that act as filters to reduce the load on downstream operations accelerating query processing. Before we discuss how PLAQUE works, we briefly discuss opportunities that can be exploited to learn predicates during query processing.

Opportunities to Learn Predicates Consider a query processing pipeline illustrated in Figure 1-a) that corresponds to the query plan generated by PostgreSQL (V 14.6) for the TPC-H query in Section 1. In this plan, $\bowtie_{l\_orderkey=o\_orderkey}$ is implemented as a hash join and nested loop join is used for $\bowtie_{l\_extendedprice>p\_retailprice}$. One opportunity to learn a predicate to accelerate query execution is to exploit the hash join implementation of $\bowtie_{l\_orderkey=o\_orderkey}$ as is proposed in prior works such as [15, 21]. In particular, since order is the build table and lineitem is the probe relation, once the hash table on order has been built, since all values of the join column of order are known (they have been read during the build phase), such information can be used to learn a predicate $p_1 = l\_orderkey \in p_{ok}$ where p_{ok} corresponds to all values of $o\_orderkey$ in the build table (i.e., order) as shown in Figure 1-b). p_1 can be used to filter tuples in the probe side (i.e., lineitem). p_1 in this example is a *membership* predicate which is effective in reducing the size of tuples for the downstream operators. We can alternatively implement the predicates learned from the equi join as *range* predicates, which are amenable to support index scan to bring additional I/O savings, as we will show in Section 4.

Besides exploiting the equi join (hash-join in particular) to learn filters, let us explore how other relational operators offer additional opportunities. We continue to use the example in Figure 1-b). For ease of illustration, we use small instances of lineitem, part and order tables, respectively.

Assume that after the execution of the build phase for order, during the probe phase over lineitem, a tuple (1,0.3,20) (① in Figure 1-b) rises to the join operator $\bowtie_{l\_orderkey=o\_orderkey}$, where it joins appropriate records in order and part tables and reaches the aggregate operator $Agg_{max}(l\_discount)$. At this stage, we can establish that the final query answer (i.e., $MAX(l\_discount)$) is at least 0.3, since 0.3 is the current maximum $l\_discount$ in the quantifying tuples reaching the aggregate

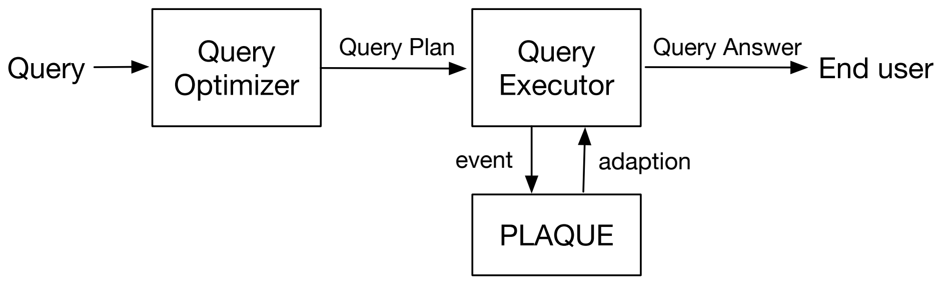


Fig. 2. PLAQUE Architecture

operator so far. We can, thus, create a new predicate p_2, i.e., $l\_discount > 0.3$, and push this predicate down to p in Figure 1-c). Such a predicate could potentially reduce the query execution cost significantly, especially in the scenario where the maximum value of the discount in lineitem is close to 0.3. In this case, all future tuples in lineitem table can be eliminated from consideration! In our current example, the second tuple ② in lineitem will be dropped away by p_2 since its $l\_discount$ (0.04) is less than 0.3.

Consider now the third tuple ③ in lineitem that joins with the record in order table to reach the nested loop join with the part table. It fails to meet the condition $l\_extendedprice > p\_retailprice$ in the theta-join for every record in part. As a result, we can learn a new predicate $p_3 = l\_extendedprice > 10$, since failure of tuple ③ with $l\_extendedprice = 10$ to join any tuple in part establishes all values of $p\_retailprice$ must be greater than or equal to 10 and thus all values in $l\_extendedprice$ must be greater than 10 in order to successfully join and produce an answer. Such a filter will allow Tuple ④ in lineitem to be eliminated since it violates p_3, which implies that it must fail the theta join operator.

Note that predicates learned above can be *refined* to more selective predicates as data processing proceeds. To see this, consider the fifth tuple ⑤ in the lineitem table. When it joins with order table and reaches $\bowtie_{l\_extendedprice>p\_retailprice}$, it fails to join any tuple in part table, and thus we can *update* p_3 to a more selective predicate as $l\_extendprice > 12$. Likewise, when tuple ⑥ in the lineitem table reaches the aggregate operator with $l\_discount$ as 0.8, we can similarly update p_2 to be a more selective predicate $l\_discount > 0.8$. The more selective predicates can prune additional tuples early further reducing query execution costs.

The example above illustrates several opportunities to learn predicates that can serve as filters to accelerate query processing from different relational operations – from equi join (p_1), theta join (p_3), and MAX/MIN (p_2). In Section 3 we consider a more comprehensive set of relational operators that can help determine predicates. We note that several predicates we learn can be refined as query processing proceeds as illustrated above - e.g., predicates learned from theta join conditions, aggregations such as MIN/MAX. Furthermore, different types of predicates can be learned from equi join conditions (e.g., range filters or membership filters), and such predicates can be implemented in different ways - as filters in memory or using an index, in which case, it could potentially reduce I/O costs of reading a relation from disk. Finally, note that the predicates learned from equi join could potentially provide more benefit if they are learned from a more downstream join operator. For instance, consider the theta join condition $l\_extendedprice > p\_retailprice$ in Figure 1-c), if we modify it to be equi join $l\_extendedprice = p\_retailprice$ and assume it uses hash join with part as the build table. The predicates learned from $p\_retailprice$ when the hash table of part is built can be pushed further down to the scan of lineitem (as part of p in Figure 1-c). Such a predicate would filter away tuples early by using the condition from a downstream join operator.

Above, we highlighted several opportunities to infer predicates during query execution that can help accelerate query execution. To our best knowledge, PLAQUE is the first such comprehensive attempt to explore learning and refining predicates during query processing to prune away redundant tuples that do not result in query results. Before we discuss PLAQUE architecture, we first highlight some key challenges that arise in learning predicates that will be addressed by PLAQUE.

Challenges. Learning predicates and using them to accelerate query execution leads to several challenges. One such challenge is to devise ways to infer and refine predicates by exploiting semantics and implementations of various relational operators that comprise a query. The predicate learned should be selective so that it prunes away as many records as possible. However, the predicate must simultaneously be correct in the sense that its usage does not change query results. Second, where should we insert the learned predicates in the query tree? As will be shown later, pushing the predicates down to the scan (leaves) of the query tree might not always be the best option. Third, it is critical to implement the learned predicates carefully such that applying them will not introduce significant overhead while ensuring the correctness of the learned predicates.

PLAQUE Design. PLAQUE addresses the above challenges by making careful design choices. The overview of the architecture of PLAQUE is in Figure 2. Given a SQL query, in PLAQUE a query optimizer first generates a query plan sent to the query executor. During query execution, PLAQUE will capture certain events to either *learn new predicates* or *update/refine* predicates that have been learned previously. PLAQUE ensures that such predicate addition and/or refinements do not change the results of the original query. Learning or refining predicates in PLAQUE are implemented using ECA rules [3] based on the state of execution of the query. An ECA rule consists of three components: an event is defined as [**WHEN:** event, **IF:** condition, **THEN** action]. As an example in Figure 1, consider the first tuple ① in the lineitem table, and the following event: [**WHEN**: tuple ① reaches the aggregate operator $Agg_{max}(l\_discount)$, **IF** tuple ① is the first tuple reaching $Agg_{max}(l\_discount)$, **THEN** a predicate $l\_discount > 0.3$] is created. Similarly, PLAQUE will detect events to learn/update new predicates from MIN/MAX, theta-join, equi-join, HAVING/GROUP BY conditions in Section 3.

Once the new predicates are learned, PLAQUE applies the learned predicates in the query executor to speed up the execution. This is achieved in PLAQUE through two subtasks:
• Deciding *where* to place the learned predicates in the given query plan tree PLAQUE makes the decision based on evaluating dependence between different query blocks of a query tree and determining a placement strategy to maximize the benefit of predicate placement (discussed in Section 5).
• Deciding *how* to implement the learned predicates in the executor, i.e., whether or not to use index-scan (discussed in Section 4).
Finally, the executor returns the query answer to end users.

PLAQUE has been implemented in an Apache project VanillaDB [8, 24], which consists of several key components (query executor and optimizer) and supports most popular operator implementations such as hash-join, index/table scan, index join, sort-merge join, nested loop join, etc. Thus VanillaDB is suitably modified for a reference implementation of PLAQUE. In particular, PLAQUE added the code to implement the learned predicates in VanillaDB by creating *in-memory* predicate or *index* predicate, which requires minimal modifications to current DBMS codes with low overhead. The in-memory predicate is implemented as an in-memory *checker* that is directly applied to the data flow among operators during query execution to eliminate any tuple that fails the corresponding predicate, while the index predicate is implemented as index-scan to fetch tuples using an index. We discuss the two implementations in detail in Section 4. Furthermore, mechanisms to add and dynamically refine predicates in the executing query using ECA rules were added to the codebase. Extending other open-source DBMSs, such as PostgreSQL, to support learned-predicate-based query execution is part of our future work.

3 PREDICATE CREATION

In this section, we describe how PLAQUE learns predicates from various relational operations in a query, including MIN/MAX aggregate, theta join, equi join, and group by/having conditions. In particular, PLAQUE aims to learn two types of predicates during query execution, i.e., *range predicate* and *membership predicate* which are of the form $[a\ op\ v]$ and $[a \in V]$ respectively, where op is a relational operator such as $>$, $\geq$, etc., and v is a value in the domain of attribute a, and V is a set of such values.

Predicates learned in PLAQUE that are used to modify the query do not result in a change of the final answers returned by the query (correctness). Furthermore, PLAQUE uses a **monotonic refinement** approach to modifying predicates learned wherein a predicate, say p may be replaced by a predicate p' learned later if p' is more selective compared to p, i.e., $p' \longrightarrow p$. As an example, a predicate $a > 10$ may be replaced by $a > 20$ since the latter is more selective. PLAQUE uses such a monotonic refinement strategy to filter more tuples thereby improving performance. Monotonic refinement of learned predicates does not jeopardize the correctness of the approach, which produces exactly the same results as that produced by the original query without learned predicates. Below we restrict ourselves to discussing only the predicate learned from different operators. Arguments about the correctness of the approach, and the exact definition of correctness in the context of adding newly learned predicates during query execution, while intuitively simple, are nonetheless, more formally treated in [5].

3.1 MIN/MAX Aggregation

Consider an aggregate query with max or min conditions on attribute a, $MAX(a)$ or $MIN(a)$. Let t be a tuple and $t.a$ be the attribute value of a in tuple t. We first describe the event that causes the corresponding ECA rule (discussed in Section 2) to trigger the creation of a predicate learned from extremal aggregate operators. We restrict our discussion to the MAX operator. The logic for MIN is very similar and follows directly from the discussion below.

Event 1. Predicate Creation from MAX Operator.
WHEN: $MAX(a)$ operator receives a tuple t
IF: t is the first tuple $MAX(a)$ receives
THEN: a predicate p, $a > \$a$, is created, where $\$a = t.a$.

Note that $a > \$a$ satisfies the predicate correctness since none of the records with values of $a \leq \$a$ would satisfy the query answer. As an example in Figure 1, consider the first tuple ① in the lineitem table. A predicate $l\_discount > 0.3$ is created when tuple ① reaches $Agg_{max}(l\_discount)$ with $l\_discount$ as 0.3. Eliminating records with $l\_discount \leq 0.3$ will not change the query results.

Once a predicate is learned from MAX aggregate operator, it may be updated later during query processing. Such a refinement is captured by the following event.

Event 2. Predicate Refinement from MAX Operator.
WHEN: $MAX(a)$ aggregate operator receives a tuple t
IF: the predicate p associated with $MAX(a)$ exists and $t.a > \$a$
THEN: update p to be $a > \$a$, where $\$a = t.a$.

The predicate refinement based on $MAX(a)$ operator defined above is *monotonic* and hence the refinement may filter additional records since the corresponding predicate is more selective. We note that predicates learned from MAX operator would be most effective if the true maximum value (or a value close to it) appears early in lineitem table, which will then allow early pruning of other tuples that would not make it pass the aggregation operator.

3.2 MIN/MAX with GROUP BY

Let us now consider MIN and MAX predicates in conjunction with GROUP BY operators. For now, let us assume there is no HAVING clause in the query which is addressed separately in Section 3.3. Let a be the attribute on which the MAX or MIN value is computed, and b is the attribute used to create groups, e.g., SELECT MAX(a), b, FROM..., WHERE..., group by b. For such a GROUP BY aggregate operator, PLAQUE adds a predicate an initial predicate p as follows at the beginning of the query processing.

Event 3. Predicate Initialization MIN/MAX GROUP BY.
WHEN: at start of query execution
THEN: Add a predicate $p = \neg(b \in \$groups)$, where $\$groups = \emptyset$.

p initially will return true for any tuple since $\$groups = \emptyset$. When a tuple t reaches the aggregation operator, the predicate p is appropriately modified by adding a new predicate p_i as a disjunct, where p_i corresponds to a predicate for the group (i.e., the b value) associated with the tuple t.

Event 4. Predicate Addition MIN/MAX GROUP BY.
WHEN: $MAX(a)$ operator receives a tuple t
IF: t is the first tuple $MAX(a)$ receives in the group whose group value $b = t.b$
THEN: create a predicate $p_i = (b = b_i) \wedge (a > \$a_i)$, where $\$a_i = t.a$. Modify the variable $\$groups$ in the predicate p associated with the aggregation to $\$groups \cup \{b_i\}$. Finally, add p_i as a disjunct to p creating a modified /extended version of p. More formally, let $p = \neg(b \in \$groups\} \vee p'$. [2] The p is modified to be: $p = \neg(b \in \{\$groups \cup \{b_i\}\} \vee p' \vee ((b = b_i) \wedge (a > \$a_i))$.

Consider a modified TPCH query in Section 1 where the aggregate attribute $a = l\_discount$ and the group attribute is $l\_shipmode = \{$ 'Air', 'Mail', ...$\}$. When the first tuple t reaches the aggregate operator whose $t.l\_shipmode =$'Air' and $t.l\_discount = 0.3$, the predicate p is updated to $\neg(b \in \{\text{'Air'}\}) \vee ((b = \text{'Air'}) \wedge (a > 0.3))$. At this time instance, if we were to apply the learned predicate p on a new tuple t' to check if t' can be skipped or not, and assume $t'.l\_shipmode =$ 'Mail', the predicate returns true and tuple t' will pass since its group does not associate with any filtering condition.

The newly learned disjunct to the predicate p associated with the GROUP BY aggregation operator contains a filtering condition $(a > \$a_i)$ which is further refined as more tuples of the same group b_i are seen as query execution proceeds.

Event 5. Predicate Refinement from MIN/MAX GROUP BY.
WHEN: $MAX(a)$ operator receives a tuple t
IF: t is in group b_i where $b_i \in \$groups$, and $t.a > \$a_i$
THEN: update p to $p = \neg(b \in \$groups) \vee p' \vee ((b = b_i) \wedge (a > \$a_i)$, where $\$a_i = t.a$.

When a new tuple t reaches the aggregate operator whose $t.l\_shipmode =$ 'Air' and $t.l\_discount = 0.8$, the predicate p is refined to $\neg(b \in \{ \text{'Air'} \}) \vee ((b = \text{'Air'}) \wedge (a > 0.8))$. For each group b_i, $\$a_i$ is the maximum value in this group observed so far during execution. In Section 4, we will detail how to implement such a disjunction of predicates.

We note that the above strategy of maintaining a predicate for each group to filter tuples may introduce non-trivial storage and processing overhead when the number of groups is large. PLAQUE uses several optimizations to reduce such overhead. To reduce the overhead of maintaining and checking a disjunction for each group, PLAQUE maintains predicates for a small set of k groups. We choose the k groups for which to maintain predicates based on estimating the size of different groups

[2]Note that after initialization, when $\neg(b \in \$groups)$, then p' is empty. As more disjuncts get added to the predicate p, the subsequent value of predicate p has a non-empty p' which itself contains one disjunct for each group that has been observed so far.

by a bootstrapping process by processing an initial sample of records without any predicates. From the sample, we determine the top-k largest groups and then subsequently learn filters on the chosen b_i values based on frequency. The intuition behind the choice is that predicate-based filtering will be most effective on such groups given their size. We can further reduce the overhead of checking if a value of b in a tuple has been previously observed (i.e., $\neg(b \in \{\$groups$) by maintaining $\$groups$ as a bloom filter. Note that false positives in the bloom filter does not jeopardize the correctness - it only implies that PLAQUE will not be able to form a predicate on b_i if the bloom filter indicates that b_i is already in $\$groups$ as a false positive.

3.3 Conditions in HAVING Clause

Consider a query with having condition, **SELECT** Agg(a), b **FROM** $R_1, \ldots, R_n$ **WHERE** . . . **Group by** b **HAVING** Agg(a) op v, where a is the aggregate attribute and b is the group attribute. op is one of $> \mid \geq \mid < \mid \leq \mid =$, v is a value, and $Agg = max \mid min \mid sum \mid count$.

During query execution, the aggregate operator maintains the aggregated value $Agg(a)$ (e.g., $SUM(a)$) for each group. $Agg(a)$ will be updated when any new tuple reaches the aggregate operator.

Consider the scenario where *Agg* is *count*, and HAVING condition is *count(a)* $<$ *100*. If the HAVING condition becomes false, i.e., *count(a)* $\geq$ *100*, it will *always* remain false during later query execution for that group. On the other hand, for the HAVING condition *count(a)* $>$ *100*, once it becomes true, it will always remain true in the future when more tuples are processed. We capture such a concept by defining *in-preserving* and *out-preserving* properties for the condition in the HAVING clause. Subsequently, we describe how to learn predicates that can be used to filter tuples based on the conditions in the HAVING clause.

Definition 1. In/Out-Preserving Property of Having Condition. A condition $H = [Agg(a)\ op\ Constant]$ in the HAVING clause is *in-preserving*, if H becomes *true* at any instance t during query execution (based on partially observed tuples belonging to a given group), H always remains *true* at any instance t' where $t' > t$, when more tuples of that group have been observed. On the other hand, H is *out-preserving*, if H is *false* at an instance t during query execution, it remains *false* at any future instance t' where $t' > t$ when more data has been observed. □

Given the above concepts of In/Out preserving conditions, we can now define the event to create the corresponding predicate.

Event 6. Predicate Learned From Having.
WHEN: $Agg(a)$ in a HAVING condition is updated for group with group value b_i
IF: the HAVING condition is out-preserving, and $Agg(a)$ fails the condition (false-condition)
THEN: a membership predicate $p_i = \neg\{b_i\}$ is created.

Whenever an out-preserving having condition becomes false during query execution in the group whose group value is b_i, PLAQUE learns the predicate p_i to skip all later tuples in the same group. In particular, for any tuple t, if $t.b = b_i$, t fails the predicate p_i and it will be skipped. Note that the In/Out-preserving property of HAVING containing *MIN*, *MAX* or *COUNT* aggregation can be decided together with op in advance of query execution. For instance, $max(a) > 100$ is in-preserving and $max(a) < 100$ is out-preserving. As for *sum* aggregate operation, if the data statistics of attribute a is known in advance, say all values in a are non-negative, then the out-preserving property of *sum* can also be determined a-priori to query execution. For instance, $sum(a) < 100$ is out-preserving if $\forall v \in Vals(a), v \geq 0$.

3.4 Learning from Theta Join

In this part, we show how to learn predicates from theta join conditions in the given query during query execution. Let R be a relation. Consider a theta join condition between relations R_1 and R_2,

$R_1 \bowtie_{a\,op\,b} R_2$, where a and b are two attributes, and $op := > \mid \geq \mid < \mid \leq$. To better illustrate the idea, without loss of generality, let us assume op is $>$, i.e., the theta join condition is $a > b$. We first define the event to trigger the creation of predicates learned from a theta join operator.

Event 7. Predicate Creation From Theta-Join Operator. **WHEN:** tuple $t \in R_1$ arrives at a theta join operator, $R_1 \bowtie_{a>b} R_2$
IF: t is the first tuple that *fails to join with any tuples* in R_2, [3]
THEN: a predicate p, $a > \$a$, is created, where $\$a = t.a$

A tuple $t \in R_1$ failing to join with any tuple in R_2 implies that for any tuple $t^{'} \in R_2$ that comes to this theta join operator, the attribute value of $t^{'}.b$ must be *greater than or equal* to $t.a$, i.e., $b \geq t.a$. Since $a > b$, this naturally implies $a > t.a$, thus establishing the correctness of the learned predicate p. Consider the tuple ③ in the lineitem table in Figure 1-b). As shown in Section 2, the theta join $\bowtie_{l\_extendedprice>p\_retailprice}$ learns the predicate $p\_retailprice \geq 10$ when tuple ③ of lineitem fails to join any tuple in the part table.

Once the predicate $p = a > \$a$ is learned, it could be updated during later query execution when $\$a$ is updated to a larger value. In particular, we define the event of predicate refinement from theta join operator below.

Event 8. Predicate Refinement From Theta-Join Operator. **WHEN:** tuple $t \in R_1$ arrives at a theta join operator, $R_1 \bowtie_{a>b} R_2$
IF: $p = a > \$a$, t fails to join with any tuples in R_2, and $t.a > \$a$
THEN: the predicate p is updated to be, $a > \$a$, where $\$a = t.a$.

The predicate refinement discussed above is monotonic. The operand $\$a$ in predicate $a > \$a$ is the maximum value of attribute a in the tuple from R_1 that failed join test in the theta join operator so far. So failure of a larger a value to join any tuple in the theta join can be used to refine the predicate to a more selective predicate while ensuring correctness of the execution. This was illustrated in Figure 1 by refining the predicate from $l\_extendedprice > 10$ to $l\_extendedprice > 12$ when processing the tuple ⑤ of the lineitem which also failed to join with any tuples in the part table in theta join operator $\bowtie_{l\_extendedprice>p\_retailprice}$.

Likewise, when op in the theta join condition is $\geq$, we follow Event 7 and Event 8 to learn exactly the same predicate as the one when op is $>$. In contrast, when op is $<$ or $\leq$, the learned predicate is $a < \$a$, where $\$a$ is the minimum value of attribute a in the tuple from R_1 that failed the join test in the theta join operator so far.

Symmetrically, for a theta join $R_1 \bowtie_{a>b} R_2$, if there is a tuple coming from the right side of the join, i.e., R_2 and it fails the join test, we create a predicate $b < \$b$, where $\$b$ is the minimum value of $R_2.b$ in the tuples from R_2 that fails join test in this theta join operator so far during query execution.

In a nested loop implementation of theta join $R_1 \bowtie_{a>b} R_2$, if a tuple rises from R_1 and the join algorithm checks the entire R_2 relation to perform the join, we refer to R_1 as outer relation and R_2 as inner relation Table 1 summarizes all the predicates that can be learned from a theta join condition based on the op and on which side the tuple t ascends into the join. In the table, we denote max_a and min_a by the maximum and minimum value in attribute a that fails join test so far during query execution. It can easily be shown that the process of replacing predicates added to the query tree earlier by stronger predicates discovered later in the execution is monotonic and ensures the correctness of the execution.

Analysis: The effectiveness of predicates learned from theta joins in accelerating query processing depends on the implementation used to implement the join. The most commonly used theta-join

[3]The above observation can be easily captured during query execution since the join output for t in current theta join operator will be empty (NULL).

Theta join	$R_1 \bowtie_{a\ op\ b} R_2$			
Outer relation	R_1		R_2	
op	$>$ or $\geq$	$<$ or $\leq$	$>$ or $\geq$	$<$ or $\leq$
Predicates	$a > max_a$	$a < min_a$	$b < min_b$	$b > max_b$

Table 1. Learned Predicates for Theta Join

implementations can be categorized as (block-based) nested loop join, index nested-loop join or sort-merge joins with a few variants, such as ripple join, that performs join $R \bowtie S$ by sampling tuples from both relations simultaneously. Consider the theta-join operator $R_1 \bowtie_{a>b} R_2$ and let the join be implemented using nested loop (or index nested loop). W.L.O.G., assume $R_1.a$ is the outer relation where each tuple t in R_1 reaches the operator, and then the operator checks t matches any tuples in R_2 (inner relation). [4] The learned predicates from such a join operator are expected to provide improvement when (block-based) nested loop join, index join and ripple join are used and the values in $R_1.a$ reaching the theta join operator is *not sorted*. The advantage of the predicate would not benefit the sort-merge implementation because tuples are processed in sort order. All the remaining tuples yet to be processed satisfy the learned predicate and, hence, would not be pruned further.

3.5 Learning from Equi Join

Equi join is the most common SQL query. We start with identifying several opportunities to learn predicates from equi join.

To identify opportunities to learn predicates from equi join we first need to define a concept of convergence point.

Definition 2. Convergence Point Let R be a relation that participates in a join in a query Q. A convergence point for R wrt to the join operator is defined to be the earliest point in the query execution when all the possible tuples of R that could possibly participate in the join have passed through the join operator at least once. □

The convergence point of a relation participating in a join depends upon the join algorithm used. For instance, in the case of a hash join, the convergence point of the build relation occurs when the corresponding hash table has been built. In the query tree in Figure 1-a), the convergence point of order table is reached after we build the hash table for order table. Similarly, the convergence point of part table is reached when the first outer loop is complete in the join operator $\bowtie_{l\_extendedprice>p\_retailprice}$ when using nested loop join. At the convergence point for a relation in a join, all possible values of the relation that could participate in the join have already been observed and this information can be exploited in learning the appropriate predicates from joins. Note that different relations reach convergence points at different instances, based on the join implementation. For instance, in Figure 3 the convergence point for lineitem occurs close to the end of query execution since lineitem is the probe table and we have to consume all tuples from lineitem table to complete the query. In general, for one-pass hash join or nested loop join, their build table or inner table will potentially reach convergence points early during query execution when the build phase is complete or the first outer loop is complete. For multi-pass hash join (e.g., grace hash join) and sort-merge join, both relations will reach their convergence points when the scan or sort for both relations is complete.

We can learn either membership or range predicates from equi joins at the convergence point of participating relations.

[4]For index join, the inner relation corresponds to the one that performs the index scan for each tuple coming from the outer relation. Ripple join switches the inner and outer relations during join execution.

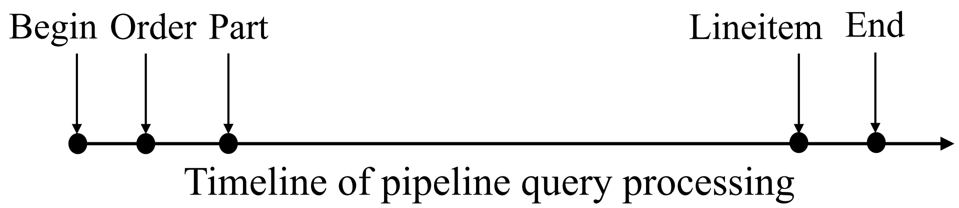


Fig. 3. Convergence Points of Relations.

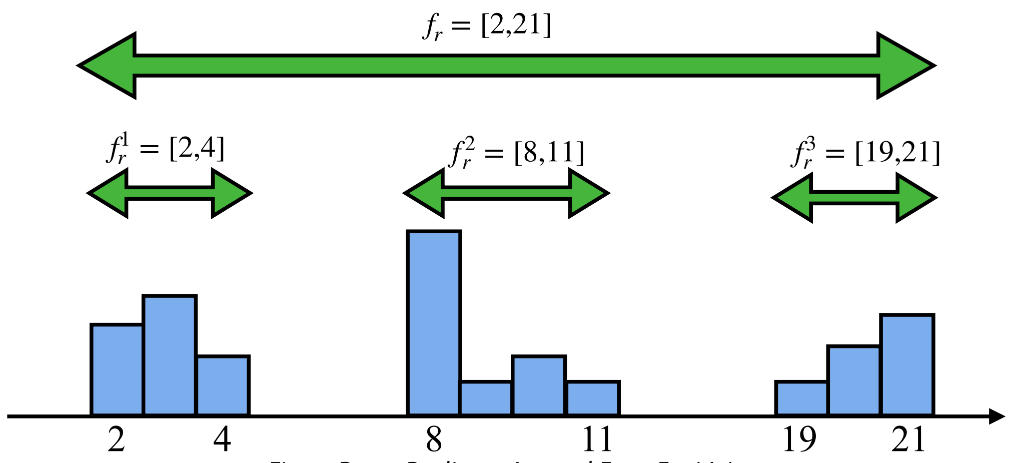


Fig. 4. Range Predicates Learned From Equi Join.

Event 9. Predicate Creation from Equi Join.
WHEN: R_1 reaches its convergence point
THEN: define a predicate p on relation R_2 on the join column $R_1.a$ where p_1 is either a set of range predicates that cover the attribute values in $R_1.a$ or p_1 is a membership predicate $a \in Vals(R_1.a)$, where $Vals(R_1.a)$ consist of all values in $R_1.a$.

We next discuss how membership/range predicates are learned.

Learning Membership Predicate from Equi Join: When a relation R reaches its convergence point early during query execution, we can learn a membership predicate p_m from the join attribute in R. We adopt the choice of bloom filters to implement membership predicates as in the previous work [15, 21, 27] to enable more efficient filtering due to the succinct nature of the bloom filters.

Membership predicates can save on the computational cost of computing join by filtering records that will not join with records in the other relation. Such functions, however, do not save the I/O cost of reading tuples from disk. For such a benefit. we instead can learn index-based range predicates. Below we describe the range predicate learning strategy used in PLAQUE that brings about 3x I/O saving compared with the membership filter as shown in Section 6). We will show how to implement an index scan using the range predicates from equi join conditions in Section 4.

Learning Range Predicate from Equi Join: Consider an equi join operator $\bowtie_{R_1.a=R_2.b}$, where a and b are join attributes in R_1 and R_2. Assume that R_1 reaches its convergence point early during query execution (e.g., R_1 is the build table) and, thus, all values of $R_1.a$ are known early at the convergence point. Our goal is to learn a set of range predicates from the values in $R_1.a$ that can be pushed down to the other relation R_2 relation (e.g., the probe side) in the query plan tree to help filter away tuples using an index on R_2, thus saving I/O costs. Figure 4 shows the histogram of values in $R_1.a$ (blue buckets) [5] where the size of a bucket is unit size, i.e., 1. Let $P_r = \{p_r^i\}$ be a set of range predicates. Our approach to learning range predicates does not explicitly construct the histogram for $R_1.a$ - we use the histogram in Figure 4 for better illustrations. To learn P_r from values in $R_1.a$ in the equi join condition $\bowtie_{R_1.a=R_2.b}$, several factors are considered.

- *Completeness*: the learned range predicates P_r should not introduce false negatives, i.e., P_r should contain all values of $R_1.a$. Otherwise using P_r on the probe side will incorrectly filter potential correct values in the query answer.
- *Effectiveness*: P_r should not result in large number false positives. One possible learned predicate for $R_1.a$ in Figure 4 is $p_r = \{[2, 21]\}$, which has zero false negatives. However, p_r is not effective since it has large false positives. (e.g., $[5, 7]$, $[12, 18]$) Instead, $P_r = \{[2, 4], [8, 11], [19, 21]\}$ may be a better set of predicates learned from $R_1.a$ since it does not introduce any false positives nor false negatives.
- *Complexity*: the number of predicates in P_r should be constrained. If we simply learn the unit predicates (i.e., create one predicate for one value, such as $[2, 2]$, $[3, 3]$, ...$[21, 21]$), P_r downgrades to a membership-like predicate but using a less efficient implementation, and $|P_r|$ will be equal to the number of distinct values in the column $R_1.a$, which increases the complexity of predicate implementation as we will show in Section 4.

The predicate p_r we learn from an equi join condition on attribute a has the format $[l, u]$, where l and u represent the lower and the upper values in attribute a. Let $l(p_r)$ be the number of domain values covered by the predicate, where $l(p_r) = u - l + 1$. For a value v, we denote by $v \in p_r$ if v is in the interval of p_r. Formally, we define the Range Predicate Learning (RPL) problem as follows.

Definition 3. Range Predicate Learning. Given a set of values V in join attribute and k, the maximum number of range predicates, RPL aims to find a set of range predicates $P_r = \{p_r^i\}$, such that,

$$\text{(RPL)} \min \sum_{p_r^i \in P_r} l(p_r^i) \tag{1}$$

$$\text{s.t. } |P_r| \leq k \tag{2}$$

$$\forall v \in V, v \in P_r \tag{3}$$

The range predicate learning defined based on Definition 3, condition 2 guarantees that the number of learned range predicates is at most k (i.e., complexity), and condition 3 makes sure the learned predicates will contain all attribute values and thus no false negatives. (i.e., completeness) By minimizing the total length of range predicates, we are able to maximize the effectiveness of the predicates since less number of false positives (i.e., gaps in the histogram of attribute values) will be introduced by more concise predicates. RPL defined above can be shown to be NP-Hard by reducing from size-constrained weighted set cover problem [11]. PLAQUE, thus, uses a fast greedy approach k-Max-Gap to find the range predicates. The detailed analysis of problem hardness and algorithm description using pseudo code is put in [5].

Example 1. Consider the example in Figure 4. Assume $k = 3$, i.e., $|P_r| \leq 3$ implying that our goal is to find at most 3 range predicates. We first sort the values in the join key and find the top-2 largest gaps (the distance between two consecutive values in a sorted list) between two consecutive values in the join key, and they are $(11, 19)$ and $(4, 8)$. Dropping these two gaps from the value interval of the join key leads to three predicates, $[2, 4]$, $[8, 11]$ and $[19, 21]$.

[5]Assume the value type of $R_1.a$ is an integer, which can easily be relaxed to float.

3.6 Sideway Information Passing

The above sections specify how PLAQUE learns predicates from relational operators. In addition, PLAQUE uses a sideway information passing (SIP) approach to learn new predicates based on predicates learned from relational operators when queries have joins.

SIP via Equi Join. Consider an equi join $\bowtie_{R_1.a=R_2.b}$. W.L.O.G., assume we have learned a predicate p_1 which is applicable in join column $R_1.a$, e.g., $p_1 = R_1.a > 10$, PLAQUE learns a new predicate p_2 in $R_2.b$ by passing p_1 via equi join condition, i.e., $p_2 = R_2.b > 10$.

SIP via Theta join. Consider a theta join $\bowtie_{R_1.a\, op\, R_2.b}$, where $op := > \mid \geq \mid < \mid \leq \mid \neq$. W.L.O.G., assume we have learned a predicate p_1 which is applicable to join column $R_1.a$. If p_1 is a membership predicate, i.e., $R_1.a \in Vals(R_1.a)$, and op is the operator $\neq$, then PLAQUE learns a new predicate p_2 on $R_2.b$ where p_2 is $R_2.b \notin Vals(R_1.a)$. Alternatively, if op is $>$, then PLAQUE can learn a predicate $R_2.b <= x$, where x is the maximum of the elements in $Vals(R_1.a)$. Similar predicates can be learned for other instantiation of the operator, e.g., if op is $<$, then we can learn the predicate $R_2.b >= x$, where x is the minimum value in $Vals(R_1.a)$.

Likewise, PLAQUE learns appropriate predicates on $R_2.b$ values based on a set of range predicates learned over $R_1.a$. Consider a predicate p_1 consisting of a set of ranges: $p_1 = p_{r1} \vee p_{r2} \vee \ldots \vee p_{rn}$, where $p_{ri} = [l_i, u_i]$ and $u_i < l_{i+1}$ learned over $R_1.a$. Based on the operator op in the theta join $\bowtie_{R_1.a\, op\, R_2.b}$, PLAQUE learns predicates on R_2 as follows. If op is $>$, then the predicate learned on $R_2.b$ corresponds to $R_2.b \leq x$, where $x = u_n$, and u_n is the largest value in the range predicates covering $R_1.a$ values. Likewise, if op is $<$, we add a predicate $R_2.b \geq x$, where $x = l_1$, where l_1 is the smallest value in the range predicates covering $R_1.a$. Note that in both the above cases, if l_1 or u_n are not bounded, we do not learn any predicate on R_2. For instance, if the first range predicate on $R_1.a$ corresponded to say $R_1.a \leq 5$, then its range is $(-\infty, 5]$. Thus, in such a situation, since l_1 is not bounded, no predicate on R_2 will result from the above join condition. Above, we have specified a few cases of how SIP predicates are learned in the case of theta joins. The comprehensive set of learned predicates depends upon the set of operators in the theta join, but the essential logic is similar to the one highlighted above.

4 PREDICATE IMPLEMENTATION

In this section, PLAQUE implements learned predicates as either *in-memory predicate* or as *index predicate.*

In-memory Predicate. The in-memory predicates can be either range predicate, membership predicate, or a disjunction of range predicates as in the MIN/MAX with GROUP BY (see Section 3.2).

Membership predicate is implemented by converting the value set to bloom filter. Note that the bloom filter will not have false negatives but may introduce false positives. Such a false positive may result in tuples going through but such tuples will be eliminated by the downstream operators, and thus will not generate wrong answers. In-memory range predicates are simply implemented as range conditions. The disjunction of range predicates learned from MIN/MAX with GROUP BY is converted into a map, where the key is the group value and the value corresponds to the filtering condition in the corresponding group.

Index Predicate. In-memory predicates are easy to implement and can be placed everywhere in the query tree. While they offer great flexibility and are able to eliminate tuples early during query execution, they do not help reduce the I/O cost of query execution. The alternate implementation of learned predicates using index can additionally offer I/O saving by exploiting index scans. Index based implementation of learned predicates is, however, more complex since refining predicates dynamically during query execution with more selective predicates, as is done in PLAQUE, becomes more complex when using an index-based implementation. (e.g., as is the case with the predicates learned from max/min aggregate operator and theta join operator). Furthermore, shifting the original scan in the given query plan tree to index scan of learned predicates at query execution time, if not carefully implemented, will generate duplicated query answers, as will be clear shortly. Index based implementation of predicates needs to be implemented carefully only when it will bring obvious performance improvement.

We thus consider implementing the index based predicate p when the following conditions are satisfied:

- index of the attribute that a learned predicate p operates on already exists in the database
- p is able to be pushed down to just above the scan of relation R that p is applicable in the query plan tree.
- the original scan of R is not index scan. (e.g., linear scan) Otherwise, the benefit from p using index scan would be diminished, and implementing index scan using more than one predicate adds high complexity to the executor, thus not worthy.
- p is a range predicate instead of a membership predicate.

We begin with a bootstrapping phase to estimate the selectivity of a learned predicate p, i.e., the percentage of the tuples satisfying p over all sampled tuples so far during a bootstrapping phase. In this stage, p is implemented as an in-memory predicate and placed in the optimal location in the query plan tree using Algorithm 1. If the selectivity of p is lower than a predefined threshold (i.e., p is selective), we shift p from an in-memory predicate to an index predicate. Let T be the timestamp when the index predicate is built and operated, and $Tuples$ be a set of tuples in R that have been already processed during query execution before T. An index scan p on R typically retrieves all tuples satisfying p, which might contain a subset of $Tuples$, leading to potential duplication of query answers. PLAQUE remembers all the RIDs of the $Tuples$, and skips $Tuples$ returned by the new index scan. Especially, when the rows are accessed in the increasing order of the record id (RID) (for efficient sequential I/O) in the table scan on relation R, PLAQUE uses a more efficient strategy to prevent duplication. Let cur_RID be the RID of the row in R at the time T when index predicate p is built. We add a predicate $RID > cur\_RID$ (implemented as in-memory predicate) in p immediately to prevent the duplication of already processed rows whose RID is smaller than or equal to cur_RID.

5 PREDICATE PLACEMENT

We next discuss the strategy used in PLAQUE to place the learned predicates during query execution in a given query tree so as to maximize its benefit of filtering away spurious tuples. Predicate placement in the traditional context before query execution is relatively straightforward - typically optimizers push down predicates as far down the query tree and as close to the relational scan as possible. Interestingly, when predicates are learned mid-flight during query execution, their placement as far down the query tree as possible might not be a good strategy. For example, assume PLAQUE learns a new predicate $R.a > val$ at a given stage during execution. Assume that the relation R was part of a join condition and was designated as a build table in a hash-join. Pushing the predicate below the hash function in such a case, if the build process has already occurred by the time the predicate is learned, would not help since the hash table based on R is already built. In contrast, placing the predicate, perhaps, above the hash-join to reduce the number of tuples that reach downstream operators could still be very useful in accelerating query execution. In general, one has to be careful on where and how to place operators in the query tree, when predicates are not known apriori and are learned during query execution. Our goal in this section is to develop a strategy that maximizes the impact of the predicate by placing it at an appropriate location in the query tree.

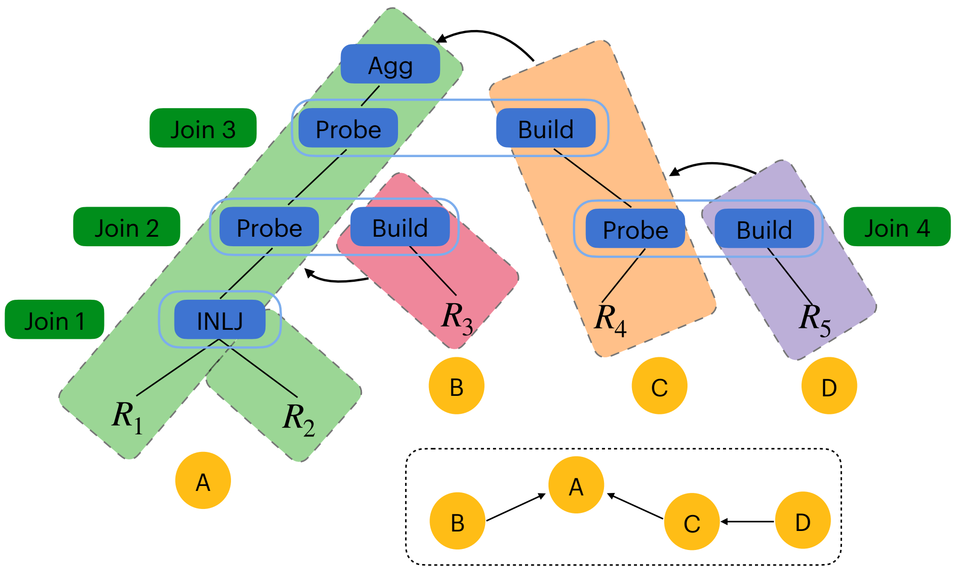


Fig. 5. Optimal Predicate Placement and Execution Graph.

In a pipeline query plan, a query is often executed in several blocks based on the specific implementations in relational operators, where all operations in one block are pipelined. Consider a four-way join aggregate query plan tree in Figure 5, where R_1 and R_2 are joined using Index Nested Loop Join (INLJ), and all the other joins are hash join. For each hash join, the probe phase is executed after the build phase is complete, leading to naturally two blocks of execution, i.e., build and probe.

Example 2. Figure 5 shows four execution blocks in the query tree, represented by nodes A, B, C, D with different colors. Let b_i be a block where all the operations can be executed using *pipelining*. In Figure 5, node A is one block where the INLJ, two probing operations, and aggregate operation can be pipelined together, while the build operation in Join 2 is in one individual block. (i.e., node B in Figure 5). Similarly, the probe phase of Join 4 can be pipelined with the build phase of Join 3 in node C, while the build of R_5 in Join 4 is an individual block.

We formulate query execution on a given query plan tree into a partial order graph to capture the order of block executions.

Definition 4. Join Graph. Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph, where each $b_i \in \mathcal{V}$ represents a block, and $e_{ij} \in \mathcal{E}$ denoting that the execution of block b_j must be executed after b_i is complete.

Example 3. The join graph in Figure 5 has four nodes, A, B, C, D. The edge from B to A denotes that all the operations in A can be executed only when the build table of R_3 is complete.

Formulating query execution as a partial order join graph is helpful in identifying where to place the learned predicates. On one hand, we wish to push the learned predicates as low as possible in the query tree to maximize their benefits to potentially skip more rows early during query execution. On the other hand, it is not beneficial to place the predicates in a block whose execution has already been completed before the time when the predicates are learned. The partial order join graph provides a way to determining where to place the predicates in the example below.

Assume a range predicate p_r is learned from the max operator (e.g., Agg in Figure 5), and it is applicable in R_1, then the best location to place p_r is just above the scan of R_1, since doing so will filter away tuples earliest. However, if p_r is applicable to R_3, inserting p_r above the scan of R_3 will not help remove tuples since the build phase of R_3 is complete before the p_r is learned in

Algorithm 1: Predicate Insertion

```
    Input: $p, G = (\mathcal{V}, \mathcal{E})$
1   $b$ := node where $p$ is created
2   $Des_G(b)$ := the set of descendants nodes of $b$ in $G$
3   for $b_i \in Des_G(b) \cup \{b\}$ do
4       $pushdown(p, b_i)$
```

the aggregate operator. Instead, the best location for p_r is R_1 (or R_2) if R_3 joined with R_1 (or R_2). Intuitively, we wish to push the learned predicate as deep as possible in the query plan tree, while ensuring the predicate will effectively filter away tuples.

We formally describe the algorithm to place any learned predicate in the query plan tree in Algorithm 1. Given a learned predicate p and the partial order execution graph $G = (\mathcal{V}, \mathcal{E})$, we first identify the node $b \in \mathcal{V}$ (i.e., block) where p is created. (Ln.1) Second, we identify the set of descendant nodes of b in graph G, $Des_G(b)$, i.e., the set of nodes that are reachable from b. Finally, in block b and each block in $Des_G(b)$, we push down the predicate p into each such block if p is applicable in the corresponding relations in the block. (Ln.3-4)

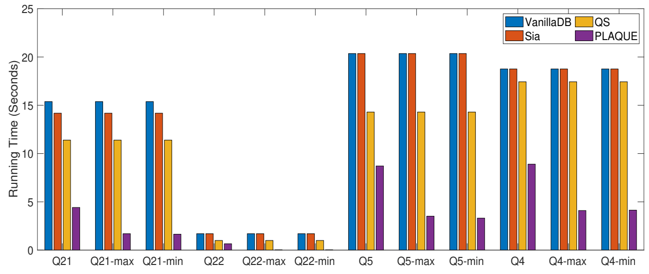
Example 4. In Figure 5, assume a predicate p is learned in node D (i.e., after the hash table of R_5 is complete). Obviously, p cannot be pushed down further in D. Consider the set of descendants of D, i.e., $\{C, B\}$. p can be pushed down in C to the probe of R_4, and it can also be immediately pushed down above the scan of the applicable base relation in node A via Join 3 (e.g., if Join 3 is $R_1 \bowtie R_4$, then p can be pushed down above the scan of R_1). Similarly, a predicate p learned in node C (after the build table of R_4 is complete) can be passed through Join 3 and pushed down to node A. Note that p learned in C will not help eliminate tuples in its ancestor nodes in the graph G, such as D, since p is learned after the block D is fully executed. When the predicate p is learned from Agg in node A (e.g., max or min operator), the only node we can push p down is A since A does not have any child nodes in the graph. In particular, p can be pushed down appropriately into the different execution points in A based on the applicable attribute of p. For instance, if p is applicable in either R_1 or R_2, then p can be pushed down to the scan of R_1 or R_2. If p is applicable in R_3, then p will be placed in probe phase in Join 2 in the node A, which is not the deepest location in A, but will help eliminate tuples early for downstream operations in A, such as the probe phase in Join 3 and aggregation.

6 EVALUATION

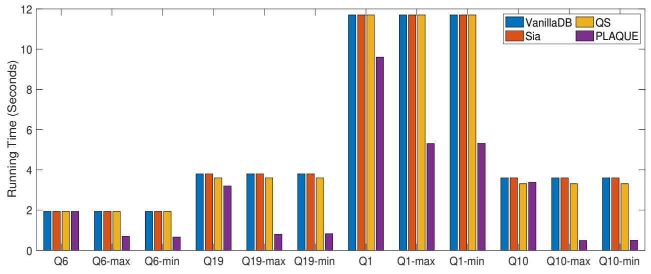
We evaluate PLAQUE's ability to accelerate query execution using two synthetic data sets and one real data set.

6.1 Methodology

6.1.1 Data Sets. • TPC-H. We use TPC-H (SF=1) as our first data set generated using standard datagen [7] which creates the uniformly distributed data. The default TPC-H does not represent a practical data distribution, which is often skewed. Therefore, we further use a modified datagen [1] to create TPC-H datasets with different amounts of skew, i.e., Zipf factor as 1 and 2, respectively.
• SmartBench. To evaluate the learned predicates on a User-Defined-Functions (UDFs) benchmark where UDFs are used in queries, we choose SmartBench [13] which is derived from a smart space sensor system and focuses on analytics of IoT data. Smartbench contains multiple sensor tables, such as Bluetooth, WiFi, or camera as well as a space table (that connects sensors to locations), where several UDFs are supported, such as location and occupancy computations.
• IMDB. We finally use a real data set IMDB [2], which contains around 4GB size files.



(a) TPC Queries (Top-4 Performance).



(b) TPC Queries (Bottom-4 Performance).

Fig. 6. TPCH Queries.

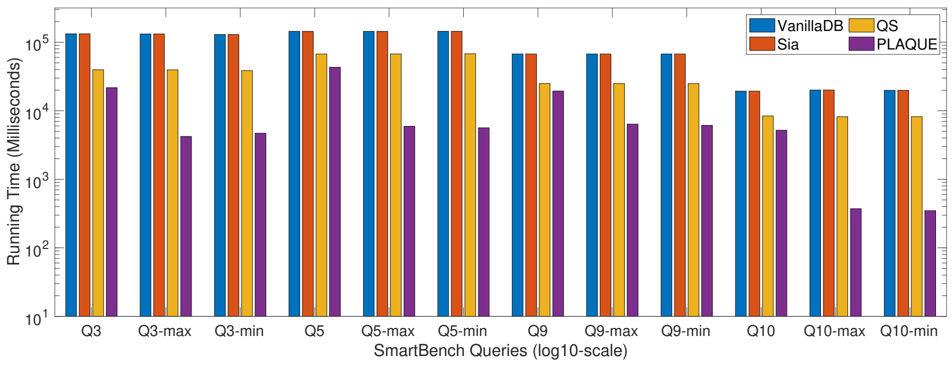


Fig. 7. Query Run Time on SmartBench.

6.1.2 Queries. We tested all queries $\{Q_1, ..., Q_{22}\}$ in TPCH. [6] In addition to testing the query Q_i to test the effect of MIN and MAX optimizations, we also test PLAQUE for Q_i with the aggregate modified to MIN and MAX, denoted by Q_i-max and Q_i-min to evaluate the predicates learned from MIN/MAX conditions. We refer to the modified TPCH query set with MIN and MAX conditions as TPCH-max and TPCH-min. In the SmartBench we pick four representative queries, $Q3$, $Q5$, $Q9$ and $Q10$, where two UDFs, computing location of a person [18] and occupancy of a room [19]. [7] In IMDB data set, we manually create four selection-projection-join-aggregate queries (i.e., $Q1$-$Q4$),

[6]For a comprehensive test, minor query modifications are made while preserving the query complexities and semantics, such as adding proper having conditions on the aggregated attributes (if any).
[7]We did not test other queries in SmartBench since they are either not interesting (without join and UDFs) or they are similar to one of the representative queries above.

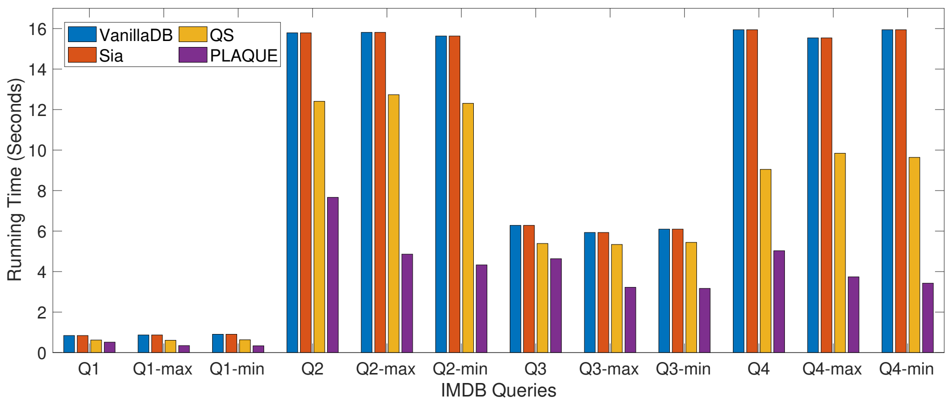
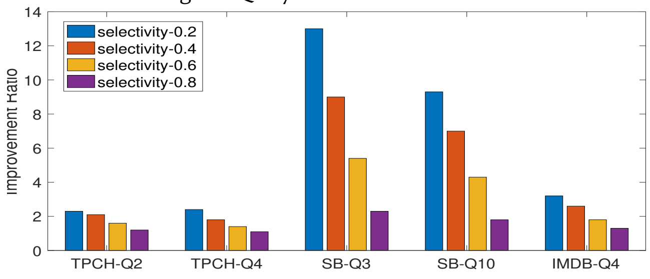
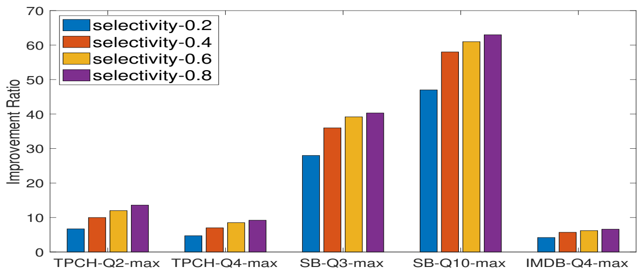


Fig. 8. Query Run Time on IMDB.



(a) Learned Predicates from Join.



(b) Learned Predicates from MAX.

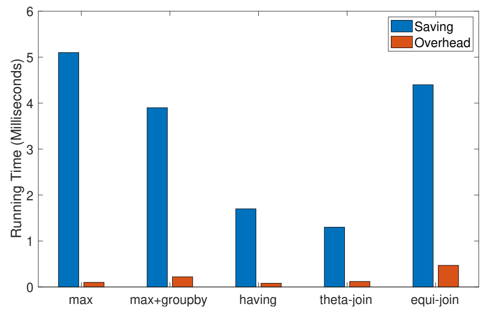
Fig. 9. Improvement Ratio on Different Selectivities.

and for each query Q_i we modify the aggregate condition to be MAX and MIN, and thus creating additional two queries Q_i-max and Q_i-min for each Q_i.

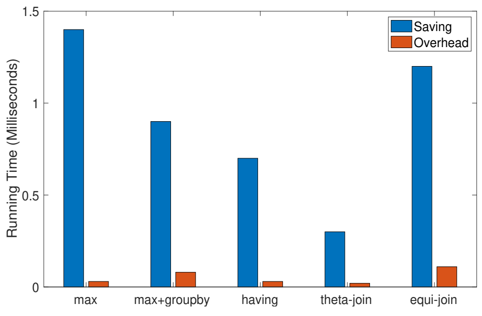
6.1.3 Compared Approaches. We compared the performance of the following four strategies. (1) **VanillaDB**: standard query optimizer and executor implemented in VanillaDB [8, 24]. (2) **Sia** [26]: Sia learns synthesized predicates given a SQL query before query execution. (3) **QuickStep (QS)** [21]: QS builds bloom filters for build table in hash join and use them to filter (4) **PLAQUE**.

6.2 Experimental Results

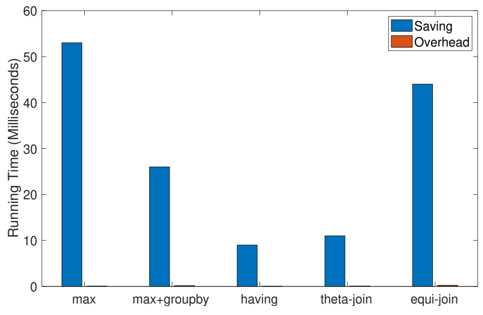
6.2.1 Performance of Learned Predicates. We start with reporting the performance of our learned predicates on TPCH (Zipf is 0 using the standard datagen [7]), SmartBench and IMDB data sets in Figure 6, Figure 7 and Figure 8, respectively. Let *improvement ratio* be $\frac{Time(VanillaDB)}{Time(PLAQUE)}$, where $Time()$ is the run time of a certain strategy.



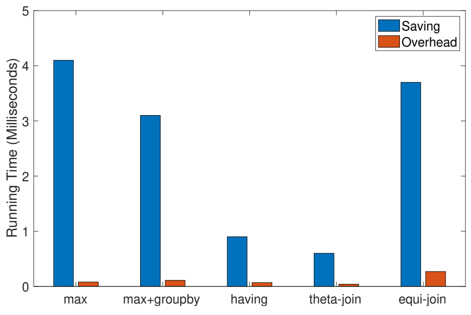
(a) TPCH-Q4



(b) TPCH-Q11



(c) SmartBench-Q5



(d) IMDB-Q4

Fig. 10. Saving and Overhead Breakdown of Learned Predicates.

Performance on TPCH. We tested all queries in TPCH and report the queries with Top-4 and Bottem-4 performance ranked by the improvement ratio of PLAQUE on the TPCH queries (not TPCH-max or TPCH-min). We report the experimental results for all TPCH queries in [5].

Figure 6(a) shows that PLAQUE, by exploiting learned predicates, achieves improvements up to 3.54x (Q21) over the basic plan without filters for TPCH queries. The improvement become up to 33.5x (Q21-min) for MIN/MAX variants of these queries. As for the bottom-4 queries in Figure 6(b), PLAQUE still shows a slight improvement over VanillaDB (and most other baselines), and a noticeable improvement is observed for their corresponding MIN/MAX queries. This is primarily because these queries contain few conditions/predicates (e.g., join, aggregate, group-by, having, etc,.) that PLAQUE could learn effective predicates from. For instance, PLAQUE has similar performance as VanillaDB in Q6, since Q6 does not have join, max/min, group-by and having conditions, and thus PLAQUE fails to learn any new predicates. However, when we add MIN or MAX conditions as in Q6-max or Q6-min, PLAQUE outperforms VanillaDB by 2.76x immediately. Overall, PLAQUE's plan with learned predicates outperforms the plan without learned predicates by 2.1x, 12.3x, and 12.7x on average in all queries in TPCH, TPCH-max, and TPCH-min, respectively. This observation demonstrates that the learned predicates could significantly speed up query execution especially when the MIN/MAX is used as the aggregate condition, the learned predicates are able to skip a large number of tuples to be processed, thus leading to significant savings.

Performance on SmartBench. In Figure 7, we use log10 scale to plot the query running time. The improvement of PLAQUE over the standard query executor, VanillaDB, is up to 6.6x for non-MIN/MAX queries and 58x for MIN/MAX queries. It demonstrates that if queries contain expensive UDFs, the impact of filtering tuples (as done by learned predicates) is even more significant.

Performance on IMDB. In IMDB data set in Figure 8, we made similar observations. The predicates learned in PLAQUE improves the standard query executor, VanillaDB, by around 2x ($Q1$) to 3.7x

($Q4$) for join queries without MIN/MAX aggregate conditions, and the improvement goes up from 2.2 ($Q3$-max) to 5.7x in $Q4$-min.

Comparison with Sia. Among all queries in TPCH benchmark, Sia is only able to generate new predicates for Q4 and Q21. For instance, in Q21, Sia leverages the condition l_shipdate < '1992-07-01' and l_shipdate > l_commitdate, thus a new predicate l_commitdate < '1992-07-01'. Sia fails to learn new predicates for all queries in SmartBench and IMDB workloads. Sia works well when the query when queries contain additional predicates on join columns. PLAQUE works in a much wider spectrum of queries and achieves higher performance improvement. Sia is complementary to PLAQUE and the predicates learned by Sia before query execution could be combined with the one learned by PLAQUE during query run time.

Comparison with QS. QS focuses on hash join and builds the bloom filter for the hash table, which is used to filter tuples in the probe table. The filtering approach of QS is included already in PLAQUE- its counterpart is learning the membership predicate from a hash join. However, PLAQUE expands the opportunities to learn predicates in several ways - based on a much larger repertoire of operators and supports both main memory and index-based implementation of the filter. The experimental results clearly demonstrates that PLAQUE significantly outperforms QS by 5.5X, 9.4X and 2.2X in TPCH, SmartBench and IMDB, respectively.

We also performed experiments on JOB [4] (Join Order Benchmark), and we observed that PLAQUE achieves significant speed ups over VanillaDB by 5.56x on average, and outperforms the best baseline QS by 4.49x on average since JOB use *MIN* as the aggregate conditions and have a relatively large number of joins in our tests. (Detailed results are shown in [5] due to space limitation.)

6.2.2 The Effect of Query Selectivity. Figure 9 examines the performance of PLAQUE approach over the standard query executor on queries with different selectivities. We pick five queries from TPCH, SmartBench (SB for short), and IMDB data sets, i.e., TPCH-Q2, TPCH-Q4, SB-Q3, SB-Q10, and IMDB-Q4, and report the results in Figure 9(a). We also pick their corresponding MAX queries in Figure 9(b). We vary the selectivity of a query to be $0.2, 0.4, 0.6, 0.8$. A query with lower selectivity indicates it is more selective since less number of tuples are in the results. We plot the improvement ratio under various selectivities of query workloads. For join queries without MIN/MAX aggregate conditions, when queries are more selective (low selectivity value), the improvement due to PLAQUE is larger. This is because for any equi join operator o, if one of its inputs, say the left side of o, o_L, is highly selective, then PLAQUE would be able to learn selective predicates from the tuples coming to o from its left side o_L, and pass the learned selective predicates along the query plan tree using the algorithm in Section 5, leading to larger improvement.

For aggregate queries with MAX conditions in Figure 9(b), interestingly, we have made a different observation. The improvement from the learned predicates is larger when the query is less selective (higher selectivity value). This is because when a query is less selective, the tuples will probably reach the aggregate operator at an earlier time and thus the predicates can be learned earlier and updated to be more selective (due to its monotonicity) in the aggregate operator using the predicate creation algorithm in Section 3. On the other hand, when the query is less selective, the improvement brought by the predicates learned from join operators is smaller as observed in Figure 9(a). It turns out the improvement due to the predicates from MAX operator is more significant than the one learned from join operators, thus leading to an overall performance improvement with the increase of selectivity. This observation indicates that the predicates learned in MIN/MAX aggregate conditions will work better for slow queries that are less selective, which demonstrates that such learned predicates are even more suitable for long-running queries with significant overheads.

Time(PLAQUE _P)/Time(PLAQUE)	IMP_avg	IMP_min	IMP_max
TPCH (TPCH-max)	1.4 (1.5)	1 (1)	2.2 (2.4)
SmartBench (SB-max)	2.8 (3.5)	1 (1)	5.8 (7.9)
IMDB (IMDB-max)	1.6 (1.7)	1 (1)	1.9 (2.3)

Table 2. Predicate Pushdown VS Optimal Predicate Placement.

Selectivity Threshold	**0**	**0.05**	**0.1**	**0.15**	**0.2**
TPCH	1.4	1.68	2.13	2.1	2.04
TPCH-max	4.2	11.9	12.8	12.4	12.1
TPCH-min	4.1	10.8	12.3	12.6	12.2
SmartBench	3.1	3.6	4.2	4.1	4
SmartBench-max	32.3	34.1	35.9	36.1	35.6
SmartBench-min	31.2	34.2	36.1	36.3	35.9
IMDB	1.3	1.6	2.3	2.2	2
IMDB-max	1.7	2.4	3.1	3.1	2.8
IMDB-min	1.7	2.4	3.1	3.1	2.8

Table 3. Improvement Ratio of Memory Predicate VS Index Predicate.

Selectivity Threshold	**0**	**0.05**	**0.1**	**0.15**	**0.2**
TPCH	0	0.06	0.11	0.24	0.28
TPCH-max	0	0.21	0.24	0.29	0.33
TPCH-min	0	0.21	0.24	0.29	0.33
SmartBench	0	0.09	0.17	0.25	0.29
SmartBench-max	0	0.38	0.44	0.52	0.54
SmartBench-min	0	0.38	0.44	0.52	0.54
IMDB	0	0.11	0.17	0.21	0.24
IMDB-max	0	0.28	0.31	0.35	0.39
IMDB-min	0	0.28	0.31	0.35	0.39

Table 4. Percentage of Index Predicates.

6.2.3 The Effect of Optimal Predicate Placement. We examine the effect of the predicate placement algorithm in Section 5 and report the results in Table 2. In particular, we compared our placement strategy with the standard strategy that always pushes the learned predicates down to the leaf nodes of the query tree, denoted by PLAQUE _P, and we reported the improvement of PLAQUE compared with PLAQUE _P, i.e., Time(PLAQUE _P)/Time(PLAQUE). We performed the tests over all queries in TPCH, SmartBench, IMDB, and all their variants (e.g., TPCH-max, IMDB-min), and reported the average, minimum, and maximum improvement of the optimal predicate placement over the predicate pushdown strategy (denoted by *IMP_avg*, *IMP_min* and *IMP_max* in Table 2). Note that due to space limitation, we do not report TPCH-min, SmartBench-min, and IMDB-min since its performance and improvements are very similar to that for the MAX queries.

In Table 2 we observe that by using our optimal predicate placement, PLAQUE could maximize the benefit of the learned predicates by placing them in the most effective locations in the query tree, which outperforms the pushdown strategy by 1.4x, 2.8x, and 1.6x on average, and up to 2.2x, 5.8x and 1.9x in TPCH, SmartBench, and IMDB, respectively. Our strategy has a slightly better performance on the query variants with MAX aggregate conditions.

6.2.4 The Effect of Data Distributions. In this experiment, we explore the effect of data distributions on the query performance in Figure 11. In particular, we use a modified datagen [1] to create TPC-H datasets with different amounts of skew, i.e., Zipf factor as 1 and 2, respectively. The standard

	Saving	**Overhead**
TPCH	3.92	0.37
TPCH-max	7.21	0.41
TPCH-min	7.13	0.42
SmartBench	67.1	0.31
SmartBench-max	84.9	0.37
SmartBench-min	82.6	0.39
IMDB	4.91	0.49
IMDB-max	7.13	0.53
IMDB-min	7.38	0.55

Table 5. Saving VS Overhead in Seconds.

TPC-H [7] comes with a Zipf as 0, which means that the data values have a uniform distribution in each column. We report the improvement ratio of PLAQUE over the VanillaDB, and discuss the result for queries with and without MAX aggregate conditions in Figure 11.

For join queries without MIN/MAX aggregate conditions (left picture in Figure 11), the improvement due to the learned predicates becomes larger on a more skewed data set with a higher Zipf value. This is expected since using the predicates learning algorithm in Section 3.5, we are able to learn a more selective predicate from equi join conditions when values are more skewed. We have similar observations for the MAX aggregate queries. The improvement from the learned predicates slightly increases on a more skewed data set, which is primarily contributed by the predicates learned from equi join conditions, and it turns out the predicates learned from MAX aggregate condition are less sensitive to the skewness of the data set than the predicates learned from join conditions.

6.2.5 Overhead and Saving of the Learned Predicates. To understand the overheads and savings of learned predicates, we report the average (in Table 5) as well as report the breakdown of the costs and savings for each type of the learned predicates (in Figure 10) for sample queries in each of the datasets. We measure the overhead of the learned predicates by computing overheads of predicate creations and applications, and we measure the savings by the query runtime reduction due to the added predicates.

In Table 5, we observe that PLAQUE pays a minimal overall overhead to achieve 8.2x, 186x, and 8.3x savings in TPCH, SmartBench, and IMDB in return. With a slightly higher overhead in the MAX/MIN variants of the above three benchmarks, we observe even higher savings due to the MIN/MAX (group-by) predicates up to 14x, 207x, and 12x, respectively. In Figure 10 we showed the breakdown of overheads and savings for each type of learned predicate. We picked four interesting queries that contain the most query conditions/predicates that can trigger the predicate learning in PLAQUE. In particular, we examined five types of predicates, the predicates learned from *single MIN/MAX*, *MIN/MAX+group-by*, *having*, *theta-join* and *equi-join.* [8] When evaluating a certain type of learned predicate, we used the knobs to disable the creations and applications of all the other types of learned predicates. We observe that the overheads of predicates learned from equi-join and group-by are noticeably higher than other types of learned predicates since learning predicates from equi-join involves possible sorts but they need to be done *only one time*, and group-by involves the maintenance of multiple range predicates in each group. Note that most range predicates (e.g., ones learned from MIN/MAX, theta join, etc,.) are very efficient since the predicate updates and applications are simply updating the operands in the predicate during query execution. Overall the MIN/MAX related predicates and the predicates learned from equi-join provided the most savings.

[8] We slightly modified the above queries such that they contain all the conditions to trigger the learning of the above five predicates.

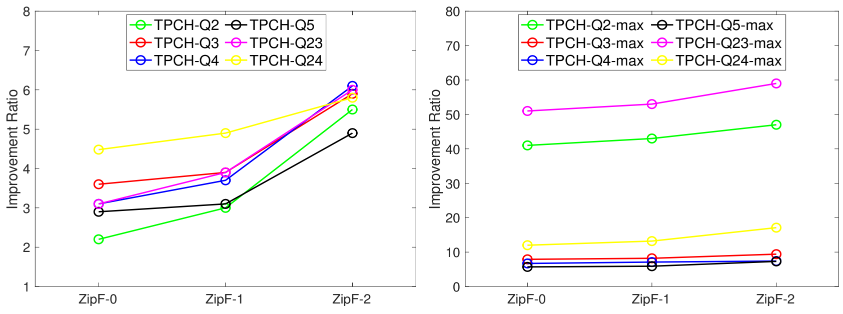
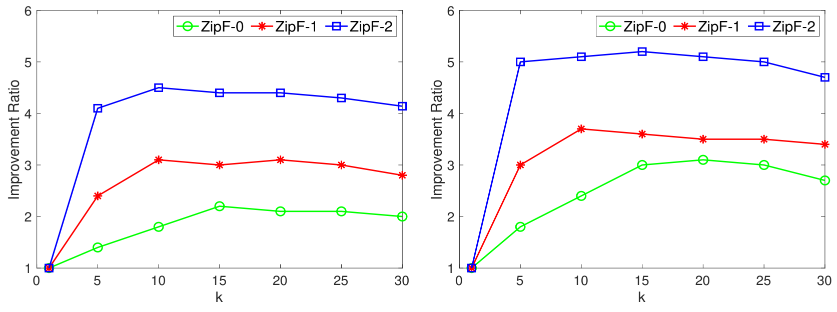


Fig. 11. Improvement Ratio on Different Data Distributions.



(a) TPCH-Q2 (b) TPCH-Q4

Fig. 12. Improvement Ratio on Various k for Predicates Learned from Equi Join.

6.2.6 *Index Predicates VS Memory Predicates.* PLAQUE implements index-based predicates when the selectivity of a learned predicate is below a given threshold ts. (see Section 4 for details) We vary ts by chosing values from $\{0, 0.05, 0.1, 0.15, 0.2\}$ and report the improvement ratio, i.e., $\frac{Time(VanillaDB)}{Time(PLAQUE)}$ and the percentage of index-based predicates in all learned predicates in Table 3 and Table 4. We observe that the best performance is achieved when the selectivity thresholds range from $[0.1, 0.15]$ in the tested datasets, and the percentage of index-based predicates accounts for roughly 20% on average. Note that the predicates learned from MIN/MAX are the most selective. When the selectivity threshold becomes larger, the time stamp of index switching tends to become later, and thus less I/O savings.

6.2.7 *Parameter Selection in Join Queries.* In Section 3.5, when we learn range predicates from equi join conditions, we use k to specify the maximum number of range predicates we wish to learn from an equi join condition. In this experiment, we explore the effect of k to the query performance on TPCH-Q2 and TPCH-Q4, by varying k from 1 to 30, and report the improvement ratio in Figure 12.

When k is 1, both queries $Q2$ and $Q4$ have the same run time as the standard query executor without improvement. In this case, one range predicate learned from equi join condition contains the maximum and minimum values on one side of the join input, which will not help eliminate any tuples, and thus will not improve the query performance. With increasing k, the improvement ratio quickly increases and then flattens out when k reaches about 10 for both $Q2$ and $Q4$. When k is too large, such as 30, the improvement ratio is slightly lower. This is because learning too many range predicates, which will although improve the selectivity of the overall learned predicates marginally, leads to additional complexity of applying the learned predicates in the query processing. Empirically, we recommend k as 10 to be the ideal setting when we learn predicates from equi join conditions.

7 RELATED WORK

The paper has already discussed (and/or experimentally studied) several approaches developed in prior related work that have also explored learning predicates [15, 16, 21, 26, 27]. Additional related work include techniques to move predicates using magic set [9, 20], algebraic equivalence [12], value-based pruning [22]. Like the syntax-driven rewrite rules in [26], this work also requires additional query specified predicates on join columns (which, as mentioned in the introduction, is not comment based on looking at standard benchmark queries, such as in TPC-H, TPC-DS).

Prior research has also explored the use of data properties, such as functional dependencies and column correlations, to accelerate query processing [10, 14, 17]. However, determining these properties can be computationally expensive (e.g., [14] employs a student t-test for each column pair). Moreover, it remains uncertain whether these properties can be sustained as data evolves. Additionally, imprecise data properties may have limited utility in query optimization (e.g., a soft functional dependency, which does not retain set multiplicity, cannot ensure the accuracy of specific plan transformations involving group-bys and joins).

Finally, we note that the predicates learned before query execution [10, 14, 16, 17, 26] (which has been the dominant line of investigation so far) are complementary to the learned predicates using our approach at query time, and they can be used together to boost query performance.

8 CONCLUSION

In this work, we study the predicate inference problem at query run time. We proposed a set of approaches to learn new predicates from aggregate, equal join, theta join, group by/having conditions, and further place the learned predicates wisely in the given query plan tree to maximize their benefit of skipping rows early during query execution, leading to possibly significant improvement. The learned filters exhibit monotonic properties, becoming increasingly selective during query processing. We have built a prototype system, PLAQUE, based on ideas described in this paper and used the implementation to conduct comprehensive evaluations on both synthetic and real datasets. Our experiments demonstrate that our learned predicates approach can accelerate query execution by up to 33x, and this improvement increases to up to 100x when User-Defined Functions (UDFs) are utilized in queries.

This work opens several interesting new research opportunities. One is to combine PLAQUE with query compilation. One could pre-compile operators with predicate templates placed at appropriate places in the query tree, and the templates can get modified/instantiated dynamically with new values as the execution proceeds and predicates are learned. The other is to explore query optimization when query processing may learn new predicates. For instance, given that PLAQUE might discover new predicates early in the nested loop join when the first tuple in the outer loop is processed, the optimizer may prefer it over other join algorithms, such as hash join in some cases, when it expects a very effective filter. Last but not least, extending PLAQUE to parallel query execution is also interesting. How to learn and share predicates in data partitions in a parallel setting, and how to optimize the data partitions based on the query predicates to learn effective predicates early are both interesting directions of further exploration.

ACKNOWLEDGMENTS

This material is based on research sponsored by HPI under Agreement No. FA8750-16-2-0021, and it is partially supported by NSF Grants No. 1527536, 1545071, 2032525, 1952247, 1528995 and 2008993.

REFERENCES

[1] 2016. *Skew TPC-H Benchmark.https://bit.ly/2wvdNVo.*
[2] 2022. *IMDB Data Set.https://developer.imdb.com/non-commercial-datasets/.*

[3] 2023. *Event Condition Action (ECA).https://en.wikipedia.org/wiki/Event_condition_action.*

[4] 2023. *Join Order Benchmark.https://github.com/gregrahn/join-order-benchmark.*

[5] 2023. *Technical Report of PLAQUE: Automated Predicate Learning at Query Time.https://drive.google.com/file/d/1QhJot-kEM9dA5TJ0rljMXFJlsh_jG03z/view?usp=drive_link.*

[6] 2023. *TPC-DS Benchmark.http://www.tpc.org/tpcds.*

[7] 2023. *TPC-H Benchmark.http://www.tpc.org/tpch.*

[8] 2023. *VanillaDB.https://www.vanilladb.org/.*

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems.* 1–15.

[10] Paul G Brown and Peter J Haas. 2003. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings 2003 VLDB Conference.* Elsevier, 668–679.

[11] Lukasz Golab, Flip Korn, Feng Li, Barna Saha, and Divesh Srivastava. 2015. Size-constrained weighted set cover. In *2015 IEEE 31st International Conference on Data Engineering.* IEEE, 879–890.

[12] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[13] Peeyush Gupta, Michael J Carey, Sharad Mehrotra, and oberto Yus. 2020. Smartbench: A benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1807–1820.

[14] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* 647–658.

[15] Zachary G Ives and Nicholas E Taylor. 2008. Sideways information passing for push-style query processing. In *2008 IEEE 24th International Conference on Data Engineering.* IEEE, 774–783.

[16] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing data-induced predicates through joins in big-data clusters. *Proceedings of the VLDB Endowment* 13, 3 (2019), 252–265.

[17] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B Zdonik. 2009. Correlation maps: A compressed access method for exploiting soft functional dependencies. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1222–1233.

[18] Yiming Lin et al. 2021. Locater: cleaning wifi connectivity datasets for semantic localization. *Proceedings of the VLDB Endowment* 3 (2021), 329 – 341.

[19] Yiming Lin, Pramod Khargonekar, Sharad Mehrotra, and Nalini Venkatasubramanian. 2021. T-cove: an exposure tracing system based on cleaning wi-fi events on organizational premises. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2783–2786.

[20] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of magic-sets in a relational database system. *ACM SIGMOD Record* 23, 2 (1994), 103–114.

[21] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment* 11, 6 (2018), 663–676.

[22] Nga Tran, Andrew Lamb, Lakshmikant Shrinivas, Sreenath Bodagala, and Jaimin Dave. 2014. The Vertica Query Optimizer: The case for specialized query optimizers. In *2014 IEEE 30th International Conference on Data Engineering.* IEEE, 1108–1119.

[23] Julian Weise, Sebastian Schmidl, and Thorsten Papenbrock. 2021. Optimized Theta-Join Processing. *BTW 2021* (2021).

[24] Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, and Yu-Shan Lin. 2012. T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD).* ACM.

[25] Xiaofei Zhang, Lei Chen, and Min Wang. 2012. Efficient multi-way theta-join processing using mapreduce. *arXiv preprint arXiv:1208.0081* (2012).

[26] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2021. Sia: Optimizing queries using learned predicates. In *Proceedings of the 2021 International Conference on Management of Data.* 2169–2181.

[27] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. 2017. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proceedings of the VLDB Endowment* 10, 8 (2017), 889–900.

Received July 2023; revised October 2023; accepted November 2023