Bridging the gap between Incremental View Maintenance and Query Plan

Qiyang He
Purdue University
West Lafayette, Indiana
he615@purdue.edu

ABSTRACT

Recently, there has been a surge in the scheduling of queries in advance to analyze the ever-growing streams of data and to reduce the latency by incremental processing. In the database area, people use incremental view maintenance (IVM) to materialize intermediate views at different levels or for distinct operators. While previous studies have attempted to build prototype systems to enhance performance, they often fail to provide convenient APIs or utilize general intermediate representations, making it challenging for developers to integrate these techniques. Furthermore, such optimizations struggle to blend seamlessly into modern database systems and often lack efficiency.

We aim to employ a query plan as the intermediate representation, bridging the gap between existing works. Furthermore, we demostarte such system can evaluate all the operators in a IVM friendly and efficient manner. Moreover, in certain specialized queries, such as aggregate nested correlated queries, our system can be tailored to accommodate even the most complex cases, including filling the holes for MAX/MIN aggregation using an innovative range binary search tree (RBST).

1 INTRODUCTION

Scheduled queries become prevailing under the increasing demand for real-time data analysis. That is, queries are placed earlier in the workflow to process on the continuously arriving streams of data. Such as executing ETL jobs or maintaining dashboard reports over dynamic datasets [39]. This type of responsive analytics play a crucial role in finance, intelligence, social media analysis and infrastructure monitor. Consider a scenario where a trader wants to know the volatility (or variance) of a stock during trading hours. Traditionally, he will issue one SQL query each time they need this information, triggering a recomputation of the entire dataset. However, the computing of variance can be decomposed into three components: the square of mean, the product between the summation of all the value and the mean, and the summation of all the square of the value. Each component can be maintained quickly, consequently, the variance can be incrementally computed efficiently. By adopting this approach, the server will preserve more computing resources for other queries as well as returning the results in real time.

The incremental view maintenance (IVM) problem statement is defined as follow: given a query Q, a database db and an update Δdb. The task is to compute $Q(db + \Delta db)$ with the previous result $Q(db)$ (+ is defined as a way of combination). Specifically, for aggregate operation `Sum` applied on attribute `A` of relation `R`. $Sum(R.A + \Delta R.A) = Sum(R.A) + Sum(\Delta R.A)$ since distributive law holds for `Sum`. Therefore, we can simply sum up the previous results and the newly delta data. However, for `Max` operation, when a deletion happens, we can hardly calculate $Max(R.A - \Delta R.A)$ by $Max(R.A)$ and $Max(\Delta R.A)$. For `join` operation, similar to the aggregate, it depends on the join conditions.

In relational database management system (RDBMS), efficiently materializing a view under updates remains a challenge. To support tuple-level views refresh, Postgresql [2] creates an additional table to store the provenance of each joined tuple. For example, the ID of R_1 is 101, the ID of T_1 is 201 and the ID of their join result R_1T_1 is 301. Then, there will be a tuple $(301, 101, 201)$ stored in an additional temporary table to represent the join relationship. (R_1, T_1 are tuples from relation R and T). This approach allows delta results to be inserted or deleted easily but at the expense of significant extra memory space and time cost. Moreover, it does not support aggregate operations yet. Oracle Database supports IVM through the use of materialized views and materialized view logs [4]. When a materialized view is created with the "fast refresh" option, Oracle tracks changes to the base tables using materialized view logs. Upon refreshing the materialized view, Oracle applies the changes recorded in the logs incrementally. So you can image, materialized view logs consume additional storage, which can become significant for large tables with frequent updates. Additionally, such lazy update method still cause latency, especially when dealing with high transaction volumes or complex materialized views. What is more, sub-query, outer joins, or certain types of aggregation functions are not eligible for fast refresh. As for SQL Server, it supports IVM through the use of indexed views. Indexed views are materialized views that have a unique clustered index on them. When the base data changes, SQL Server maintains the indexed view incrementally by updating the affected rows in the view. With the same limitation as Oracle, SQl server also suffer from large write overhead.

Some previous database works are proposed to handle some limitation above. Non-distributive aggregation functions were investigated by Themistoklis et al [30]. SB-tree was proposed by Jun and Jennifer [40] to deal with temporal aggregates. Next, Larson's and Zhou's algorithm [22] introduced an efficient maintenance procedure to cover the outer-join.

Following the idea of creating delta queries [30], DBToaster [20, 21] proposes the *High-Order IVM* (HO-IVM) where delta queries will be maintained recursively to help maintain the original query. Just like using the second order derivative to get the first order derivative and then use the first order derivative to calculate the orginal function. The use of higher-order IVM makes DBToaster the state-of-the-art incremental query processing system and shows substantial speedup over some commercial database systems [20].

In order to apply derivation more easily, a *generailized multiset relations* (GMRs) based query language, *AGgregate CAlculus* (AGCA) is proposed in DBToaster [21] to represent a query based on the aggregate `Sum`. Based on that, `Count` and `Average` can be easily expressed. (`Count` equals summation for a all one attribute, `Average` is the division of `Sum` and `Count`). But there is no specific syntax for universal quantification or aggregate like `Max`, `Min` and `Sort`. Although nested sum-aggregate queries can express these missing features, it not only complicates the query representation but also hinders maintenance efficiency. As discussed in [6], correlated nested aggregate queries do not meet the requirement that delta queries must be simpler than the higher-level query. Additionally, HO-IVM requires identifying a set of sub-queries to materialize rather than materializing a single full view, which significantly increases the cost for the compiler to find an optimal division. Using DBToaster outside the scope of IVM can also be challenging due to the basic data model GMRs, which makes efficient processing difficult since an underlying map must be used. The underlying multiset relation forces the constant maintenance of a hashmap for aggregating all results. Consequently, while DBToaster expands the generality of incremental maintenance to a large extent, it misses some operators, has a costly compiling process, and is not as useful outside of IVM.

Query plans, as the most popular intermediate representation in modern database systems, connect end-to-end processes from query parsing and optimization to execution, encompassing cost estimation, concurrency control, and debugging. Given their powerful representation, most optimization techniques mentioned in HO-IVM can be expressed through query plans. Therefore, integrating incremental maintenance techniques into query plan specialization becomes crucial for effectively implementing IVM, especially since most companies have supported IVM at an early stage. Reusing existing frameworks is preferable in order to implement this feature comprehensively.

In [6], the author proposes a general incrementalization algorithm to generate IVM implementation for nested aggregate correlated queries, as well as a specific optimization for certain classes of queries with multiple conjunctive equality predicates or a single inequality predicate. The time complexity for maintenance is $O(n)$ and $O(n \log n)$ respectively. Based on their target query representation, we can abstract the corresponding query operator pattern. The size of the pattern is not fixed, which provides the compiler with numerous choices and makes the optimization process costly. Additionally, the query requirements are quite strict. If the optimization algorithm fails to find the patterns, the performance reverts to the general method. The issue is that it focuses only on specific queries that can be fully optimized, neglecting those that can be partially optimized. Despite these limitations, it is still a pioneering effort to integrate query plans and IVM.

In this report, we explore some challenges that are in the way between query plan the IVM and investigate the query plan based IVM compiler.

(1) We present a case study of nested aggregate queries and analyze why it cannot be supported by existing approaches. Then, we motivate our approach of building IVM compiler on query plan (section 2).
(2) We re-structure the query evaluator for IVM by converting the data-centic model to a push model (section 3). Next, we present the IVM evaluation strategy for common operators (section 4) and how to extend its generality (section 8).
(3) We introduce the `GroupJoin` to show its power of accommodate most existing algorithm (section 5). Furthermore, we proposed new algorithm to bridge the gap between existing method with the `MAX/MIN`-based aggregate nested query (section 6).
(4) We present a novel binary search tree based data structures to fit the algorithm mentioned above (section 7).

2 MOTIVATION

In this section, let us recap the representation of *AGgregate CAlculus* (AGCA) proposed by [20]. Next, we will see how to fit it into a view maintenance friendly program step by step based on its logical plan.

First of all, let us have a look on the AGCA proposed by DBToaster [20, 21]. Assume c denotes the constants, x is attributes of a relation, $\vec{t}$ represents tuples of attributes, R symbols relation names, $theta$ means all the comparison operator ($>, <, =, \neq, \geq$ and $\leq$), and $:=$ is used for assignment. Additionally, with the bag union $+$, natural join $*$, and aggregate sum $\text{Sum}_{\vec{A}}$ (group by tuples of variables $\vec{A}$). The abstract syntax of AGCA is as follow:

$$q := q * q \mid q + q \mid -q \mid c \mid x \mid R(\vec{t}) \mid \text{Sum}_{\vec{A}}(q) \mid x\ \theta\ 0 \mid x := q.$$

The AGCA is closed under update deltas, that is to say, we can use the ΔQ in the same language to describe the results of Q changes when the change workload ΔD is applied in the database D:

$$\Delta Q(D, \Delta D) := Q(D + \Delta D) - Q(D).$$

Since the strong compositionality for the language, we can apply the following rules again and again until we can the basic GMRs or delta GMRs (update). We use u to represent the update, and $\Delta_u Q$ to denote the update of query Q.

$$\begin{aligned}
\Delta_u(Q_1 + Q_2) &:= (\Delta_u Q_1) + (\Delta_u Q_2) \\
\Delta_u(Q_1 * Q_2) &:= ((\Delta_u Q_1) * Q_2) + ((\Delta_u Q_2) * Q_1) \\
&\quad + ((\Delta_u Q_1) * (\Delta_u Q_2)) \\
\Delta_u(-Q) &:= -\Delta_u Q \\
\Delta_u c &:= 0 \\
\Delta_u x &:= 0 \\
\Delta_u(x\ \theta\ 0) &:= 0 \\
\Delta_u(x := Q) &:= (x := (Q + \Delta_u Q)) - (x := Q) \\
\Delta_u(\text{Sum}_{\vec{A}} Q) &:= \text{Sum}_{\vec{A}}(\Delta_u Q)
\end{aligned}$$

These rules can be regraded as a ring formed by GMRs with $+$ and $*$, and have been studied thoroughly in [19].

If we simply add the `Max` or `Min` in the syntax, these rules will be broken since the `Max` and `Min` operations are not *streamable* [16]. Thus we cannot capture the $\Delta_u(\text{Max}_{\vec{A}} Q)$ by the $\Delta_u Q$:

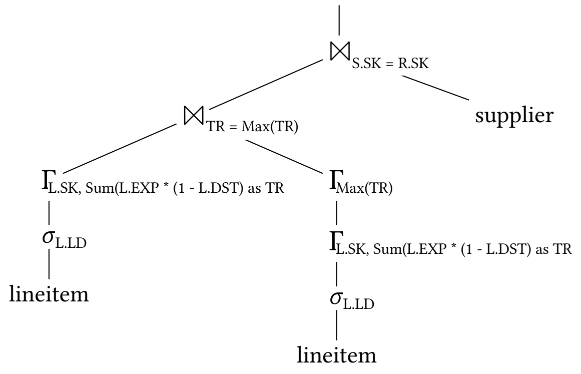
$$\Delta_u(\text{Max}_{\vec{A}} Q) \not\models \text{Max}_{\vec{A}}(\Delta_u Q).$$

Furthermore, the compiler is not guaranteed to terminate after adding the syntax for `Max/Min`.

The way to express `Max/Min` or other non-streamable aggregate is using nested sum-aggregate queries. A simplified version of query

```
1 WITH Revenue AS (
2 SELECT l.SK AS SN, Sum(l.ETP * (1 - l.DST)) AS TR
3 FROM Lineitem l
4 WHERE l.SD >= date '1996-01-01' AND l.SD < date '1996-04-01'
5 GROUP BY l.SK)
6
7 SELECT s.SK, s.NA, s.AD, s.PH, r.TR
8 FROM Supplier s, Revenue r
9 WHERE s.SK = r.SK AND r.TR = (SELECT Max(TR) FROM r)
```

(a) Condensed SQL query



$\bowtie_{\text{S.SK = R.SK}}$
$\bowtie_{\text{TR = Max(TR)}}$
supplier
$\Gamma_{\text{L.SK, Sum(L.EXP * (1 - L.DST) as TR}}$
$\Gamma_{\text{Max(TR)}}$
$\sigma_{\text{L.LD}}$
lineitem
$\Gamma_{\text{L.SK, Sum(L.EXP * (1 - L.DST) as TR}}$
$\sigma_{\text{L.LD}}$
lineitem

(b) Query plan

```
Project [s.SK, s.NA, s.AD, s.PH, r.TR]
  Join Inner [s.SK = SN]
    Project [s.SK, s.NA, s.AD, s.PH]
      LogicalRDD [S.SK, s.NA, s.AD, s.PH, ...]
    Filter [isnotnull(TR) AND TR = Sub-query(MAXTR)]
      Sub-query
        Aggregate [MAX(TR) as MAXTR]
          Aggregate [l.SK] [SUM(l.ETP * (1.0 - l.DST)) AS TR]
            Project [l.SK, l.ETP, l.DST]
              Filter [l.SD >= '1996-01-01' AND l.SD <= '1996-04-01']
                LogicalRDD [l.SK, l.SD, l.ETP, l.DST, ...] FROM Lineitem l
      Aggregate [l2.SK] [l2.SK AS SN, SUM(l2.ETP * (1.0 - l2.DST)) AS TR]
        Project [l2.SK, l2.ETP, l2.DST]
          Filter [l2.SD >= '1996-01-01' AND l2.SD <= '1996-04-01']
            LogicalRDD [l2.SK, l2.SD, l2.ETP, l2.DST, ...] FROM Lineitem l2
```

(c) Optimized query plan by SparkSQL

$$\begin{aligned}
\text{R} &:= \text{Sum}_{[SK]}(TR := LI(DST, ETP) \\
&\quad * ETP * (1 - DST)) \\
\text{Q}_{15} &:= \text{Sum}_{[SK,NA,AD,PH,TR]}(S(SK, NA, AD, PH) \\
&\quad * R(SK, TR) * (v_1 := \text{Sum}_{[]}(R(SK', TR') \\
&\quad * (TR' > TR)) * (v_1 = 0))
\end{aligned}$$

(d) AGCA expressions

Figure 1: SQL query, query plan, optimized query plan and AGCA expression for simplified query 15 in TPC-H dataset [5].

15 in TPC-H [5] dataset will be used as an example of a nested aggregate query. The SQL query, query plan, and the AGCA expression of this query which contains `Max` in its sub-query are shown in figure 1. The `Order_by` and `Sort` is ignored since it is out of the scope of this paper. To represent query precisely, the condensed schema $s(SK, NA, AD, PH)$ (s for supplier, SK for suppkey, NA for name, AD for address, PH for phone), $l(SK, ETP, DST, SD)$ (l for lineitem, SK for suppkey, ETP for extenedprice, DST for discount, SD for shipdate) and TR for total_revenue will be used.

In figure 1d, R means the first materialized view in figure 1a (line 1-4), and Q_{15} covers the main query in figure 1a (line 6-8). In Q_{15}, we use a nested sub-query v_1 to calculate the number of item in R whose TR is larger than the TR from the outer query. Furthermore, the condition ($v_1 = 0$) forces that there is no larger TR. Combined together, they make up the logic for `Max`.

Since AGCA does not support `Max`/`Min` operators inherently so the query 15 was written by nested query in [21] to make it runnable on the DBToaster engine. Using nested sum-aggregate correlated queries to support `Max` makes the representation quite complex based on the syntax of AGCA. Additionally, as explored in [6], we know that maintaining the view for nested correlated aggregate queries is the weakness of DBToaster, so the solution impairs the efficiency of view maintenance further.

To fix this hole, some data structure-based solutions as range indices are feasible [21]. As introduced in [6], parent-relative hash-based (PAI) and tree-based (RPAI) indices are designed to optimize the efficiency for nested aggregate correlated queries, which are also referable to reduce the time cost when `Max`/`Min` operators are evaluated to nested sum-aggregate queries. The target queries of PAI and RPAI are in the form of:

$$\text{AggrQ}_{[cols]}(\text{AggrFunc}, R_1, \cdots, R_n, v_1\theta q_{R_1} \cdots\ AND\ v_n\theta q_{R_n})$$

This form is completely different from the AGCA expression. Thus, we can apply some pattern matching algorithm to find some sub-queries fitting such forms and incrementalize the respective portion while using the DBToaster to handle the rest of the query. Two issues will be arisen: (1) Large amounts of transformation between these two representations needs to be done which will raise the time cost as well. (2) The number of the relations involved in this form is not fixed, which means for a join involving k relations, there will be k different sub-queries can be handled by PAI or RPAI. Following this way, we can hardly decide where to stop, whether we will apply RPAI or PAI again. A specialized optimizer is also essential for applying these indices.

Therefore, is there any representation general and can cover all the optimization above? From the AGCA representation in figure 1d, we can find that Sum accords to the `Aggregate` operator, $*$ accords to the `Join` operator and other condition can be filled in

```
class HashJoin(left: Op, right: Op)
(lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent = null
  def open() = {
    left.parent = this; ri ght.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce()
    isLeft = false; right.produce()
  }
  def consume(rec: Record) = {
    if (isLeft) hm += (lkey(rec), rec)
    else
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class IVMHashJoin(left: IVMOp, right: IVMOp)
(lkey: KeyFun)(rkey: KeyFun) extends IVMOp {
  val hmLeft =  new HashMultiMap()
  val hmRight = new HashMultiMap()
  var parent = null
  def open() = {
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def consumeLeft(rec: Record) = {
    for (rr <- hmRight(lkey(rec)))
      parent.consume(merge(rec, rr))
    hmLeft += (lkey(rec), rec)
  }
  def consumeRight(rec: Record) = {
    for (lr <- hmLeft(rkey(rec)))
      parent.consume(merge(lr, rec))
    hmRight += (lkey(rec), rec)
  }
}
```

(b)

Figure 2: `HashJoin` Implementation (a) the data-centric model and (b) IVM model

`Select` operator. Thus, we can use the query plan to represent the AGCA. For the pattern proposed by RPAI paper [6], it is also a combination of `Aggregate` and `Join` (you can refer to the section 5 in detail), thus query plan is also a suitable representation for this optimization. Therefore, we will use the query plan to build a query compiler for IVM. The details are shown in the following sections.

3 STRUCTURING THE IVM EVALUATORS

As discussed in [34], there are two kinds of mainstream query evaluation models in database, the iterator (Volcano) model and the data-centric (produce/consume) model. The volcano model is based on the `next()` interface with the *pull-based* operation. While the data-centric model is based on the `produce/consume` interface with the *push-based* operation. The iterator model is more intuitive but the data-centric model leads to more efficient compilation results as improving data and code locality.

If we think about the data flow in IVM, we will find that, even if it can also be represented by a query plan, one update will always trigger a series of updates along a path from a leaf node to the root. In the volcano model, the `next()` operation (from the root operator) will invoke recursively all the `next()` until a scan (or materialized state) is reached. The invoking order in the data-centric model is almost the same. However, for the IVM, the data flow does not happen top-down (one parent always invokes the interface from its successor nodes). Concretely, take the `HashJoin` with two scan operators from relation `A` and `B` joining on the attribute `t` as an example as shown in figure 2. The code of data-centric model is shown in figure 2a, the `produce` method in the `Join` will invoke the `produces` from `A` and `B`. Thus, it leads to better locality for the data compared to the volcano model. However, for the IVM, data will come one by one, a fine-grained update is needed. So the data-centric model cannot improve higher data locality here. The implementation for the `HashJoin` for the IVM is in figure 2b, as you can see, each `consume` will update the intermediate views (or indexes) in the current operator and then invoke the `consume` of its parent node and repeat this recursively until the root is reached.

As for the data-centric evaluation with callbacks proposed by the LB2 [34], it does simplify the state for the operators. But replacing the `produce` and `consume` with the `exec` and callback function limit the flexibility if some node is used twice in one query plan. It always happens for the scan operator. The query plan used by the existing database is a tree-like structure. The scan operator for the same relation will appear more than once in the query plan (different parents cannot share the same child in a tree). As for the fixed callback function directly from the parent from LB2 [34], since it is used in single-pass compiler, all the logic for the evaluation will be determined along the way the `exec` is invoked from the root operator to the scan operator. Thus, it will be very difficult to change or merge the callback function delivered to one operator. With using such idea for the IVM, we will duplicate the part for the same scan operator leading to a unnecessary cost for evaluation.

Up to now, we have seen the drawbacks of using the data-centric evaluation for the IVM. But maintaining the links between parent and the child node is still what we want to keep since the new query plan for the IVM will be a directed acyclic graph (DAG). Such method make sharing the output of one operator (usually the scan operator) easier. Following the idea from figure 2b, we can see that the `consume` from the scan operator or a materialized state will invoke all the `consume` from its ancestors' `consume` so that the materialized view along the query plan will be updated with the new update.

Unlike the common query execution in the traditional database, we want to maintain the view. Thus, we need to maintain more data structures (indexes) to keep some intermediate states. As for the `HashJoin` in the figure 2, there is only one hashmap used in data-centric model (shown in figure 2a) while we need to maintain two hashmaps for both relations for IVM. The case for the `HashJoin` is not that hard since all the updates are insertion or deletion, but

```
SELECT
  SUM(l.ETP) / 7.0 as AVG_YEARLY
FROM
  Lineitem l, Part p
WHERE
  p.PK = l.PK AND
  p.BD = 'Brand#23' AND
  p.CT = 'MED BOX' AND
  l.QTY < (
    SELECT
      0.2 * AVG(l2.QTY)
    FROM
      Lineitem l2
    WHERE
      l2.PK = p.PK)
```

(a) Condensed SQL query

```
1  Aggregate [SUM(l.ETP / 7.0) AS AVG_YEARLY]
2    Project [l.ETP]
3      Join Inner [l.QTY < 0.2 * AVG(l2.QTY) AND l2.PK = p.PK]
4        Project [l.QTY, l.ETP, p.PK]
5          Join Inner [p.PK = l.PK]
6            Project [l.PK, l.QTY, l.ETP]
7              LogicalRDD [l.PK, l.QTY, l.ETP, ...] FROM Lineitem l
8            Project [p.PK]
9              Filter [p.BD = 'Brand#23' AND p.CT = 'MED BOX']
10               LogicalRDD [p.PK, p.BD, p.CT, ...] FROM Part p
11       Filter [isnotnull(0.2 * AVG(l2.QTY))]
12         Aggregate [l2.PK] [0.2 * AVG(l2.QTY), l2.PK]
13           Project [l2.PK, l2.QTY]
14             LogicalRDD [l2.PK, l2.QTY. ...] FROM Lineitem l2
```

(b) Optimized query plan generated by SparkSQL

Figure 3: SQL query, query plan by SparkSQL for query 17 in TPC-H dataset [5].

for the Select or ThetaJoin where one side is a aggregate value, the maintenance becomes much more difficult. Since there is no insertion or deletion for one tuple, the incoming data will update the aggregate value, then lead to a different filter. We will discuss it later.

Another problem is that there may be duplicate indexes if we use the IVM model. Specifically, we have three relations A, B and C and there are two HashJoin (equi-join): A joins B at the attribute t; B joins C also at the attribute t. Thus, according to our IVM model, we will generate two same indexes for the relation B at the attribute t since they are from completely different HashJoin operators. Therefore, a initial idea for the IVM compiler is storing all the indexes in an *index manager* globally so that whenever one operator needs to construct a new index, it will firstly check whether we have already built it, if yes, it will re-use it; otherwise, a new index will be built. The details and more effective ways are left for the readers.

4 EVALUATION STRATEGY FOR COMMON OPERATORS

As discussed in the previous section, we will add more indexes for all the operators to maintain all the views faster. Let us have a look on a query plan for query 17 generated by the SparkSQL [8] shown in figure 3. Query 17 is a aggregate query with a correlated sub-query.

Similar as the way to represent the query 15 in figure 1, the Order_by and Sort is ignored since it is out of the scope of this paper. To represent query precisely, the condensed attributes (*PK* for partkey, *DB* for brand, *CT* for container in relation Part), (*PK* for partkey, *ETP* for extenedprice in relation Lineitem) are used. To avoid writing all the attributes in these two relations, I use the ellipsis to represent unnecessary attributes in the LogicalRDD used in figure 3b. The query 17 depicts the summation of the extendedprice for those lineitems whose quantity is less than one fifth of the average quantity sharing the same partkey and satisfying some requirements for brand and container.

As you can see from the query plan generated by the SparkSQL in figure 3b, there are so many Project and Filter operators used to link those Join and Aggregate. If we still stick to adding index for each operators, it will cost too much space and make the view maintenance time-consuming. Thus, there will be no auxiliary index for the Project operator. For example, in the line 13-14 of figure 3b, whenever, there is a new tuple inserted to the relation Lineitem, Project will apply a projection on that tuple and then deliver to the next Aggregate operator in line 12. In this way, the Project will be very lightweight since it only needs to store a hash function with several attributes. For the Filter (born from the Select operator), we can apply the same idea only storing the filter condition.

However, the case becomes much more complex if there exists a sub-query, let us recall the example for query 15 shown in figure 1. As you can see, there is a sub-query to get the maximum total_revenue (TR for short). The figure 1c shows the optimized query plan by SparkSQL. To deal with the sub-query, SparkSQL duplicates the calculation for the TR (line 8-11 and line 12-15 in figure 1c). Then, it uses the MAXTR as a filter in line 5. The difference between this Filter with the Filter in query 17 (figure 3b is that the value used here is a sub-query which means that the value is not fixed but will update with incoming tuples. In SparkSQL [8], the sub-query is treated as a expression, instead of an operator in the condition of the Filter operator. By this way, the update from the sub-query can hardly be transitioned to the Filter. To mitigate the inconvenience, we can unnest the sub-query, converting it to be a Join with the same condition and reducing the sub-query layer. Similar to the first half condition in line 3 of figure 3b. There is still some work to be done to make the symbol of attributed consistent when unnesting. There are some previous work concentrating on unnesting from SQL Server [11, 13] and Hyper [29].

After discussing the IVM implementation for Filter and Project, we can focus on Join and Aggregate right now. As covered in the section 3, the IVM implementation for the inner equiv-join, there are two hashmaps instead of one to maintain both relations. Then, for the full outer, left outer or right outer joins, we can apply similar ideas to create two hashmaps to maintain efficiently. Following the same idea, we can use two hashmaps to exclude tuples for the left or right anti-join.

When it comes to the `Aggregate`, things are almost the same as in common query execution. For aggregate with group-by (line 12 in figure 3b), there will be a hashmap to keep the aggregate value with the corresponding group keys. For aggregate without group-by (line 1 in figure 3b), we will only create one variable to monitor the aggregate changes.

5 GROUP-JOIN

Previously, we have seen the evaluation idea for each operator individually. In this section, we will explore the possibility of combining the multiple operators mentioned above to optimize the IVM evaluation strategy.

As proposed by RPAI [6], correlated nest aggregate queries are not handled efficiently by DBToaster, so the PAI and RPAI are proposed to optimize it with equalities and inequalities respectively. Specifically, it uses a template to show the queries they can optimize:

$$AggrQ_{[cols]}(AggrFunc, R_1, R_2, v_1\ \theta\ q_1)$$

where $AggrQ$ represents it should be an aggregate query, $cols$ are the attributes group-by applied to, $AggrFunc$ means the aggregate function, R_1 and R_2 are the relations involved in, and the $v_1\ \theta\ q_1$ denotes the join condition. The v_1 is either a $const$ or an $AggrQ$ without free variables so that it can be maintained independently. The q_1 is another $AggrQ$.

Let us see how to accommodate such optimization in the query plan. Previous work [25] proposed the `Groupjoin` to optimize the query evaluation, specifically, it consists of a `Join` with an `Aggregate`. As you can see the line 1 3 in the figure 3b (we can ignore the `Project` used to link the other two operators), a `Groupjoin` can be used to replace these 3 operators. The `Aggregate` operator shows the $AggrQ_{[cols]}(AggrFunc)$ and the `Join` operator represents the R_1, R_2 and $v_1\ \theta\ q_1$ after. Thus, we have seen how to apply the RPAI and PAI optimization based on the query plan for two relations. To extend it to contain multiple predicates, RPAI paper represents it in this way:

$$AggrQ_{[cols]}(AggrFunc, R_1, \ldots, R_n, v_1\ \theta\ q_{R_1}\ \ldots\ AND\ v_n\ \theta\ q_{R_{n-1}})$$

where there are n relations involved in this huge `Groupjoin`. Correspondingly, we can add more `Join` operators below the pattern (line 1 3 in figure 3b) we mentioned above to match this optimization strategy. After mining the pattern, we can apply similar optimization as in RPAI and PAI.

6 INCREMENTALIZATION OF NESTED MAX/MIN

Even if RPAI and PAI [6] can be used to optimize `Groupjoin`, it is limited by the aggregate `Sum`, it does not work for `Max/Min`. We will discuss about the difficulties of incrementalization of nested `Max/Min` in this section. Without loss of generality, we will only use `Max` in the example. The case for `MIN` can be done in a similar way. Following the same idea of [6], we will also use two examples to show the restrictions of existing system and propose new ideas about the indices to efficiently support nested `Max/Min`.

6.1 Nested Max with Equality

Example 6.1. *This query is from the example 2.1 in [6], which computes the aggregate sum of A and B from the relation R for those satisfying a* `Sum`*-based equality predicate.*

```
Q = SELECT Sum(r.A * r.B) FROM R r
WHERE                             ┌─lhs_sum
  0.5 * (SELECT Sum(r1.B) FROM R r1) =    ┌──rhs_sum
      (SELECT Sum(r2.B) FROM R r2 WHERE r2.A = r.A)
```

The following query changes the `Sum` to `Max` in the nested subquery.

Example 6.2. *This query uses the* `Max` *in the predicate of the sub-query comparing to the example 6.1.*

```
SELECT Sum(r.A * r.B) FROM R r
WHERE
0.75 * (SELECT Sum(r1.B) FROM R r1) =
(SELECT Max(r2.B) FROM R r2 WHERE r2.A = r.A)
```

These queries only involve one relation $R(A,\ B)$. The outer query is about the summation of A and B. The inner query consists of two sides, the left one is a non-correlated summation while the right one is a correlated `Max` query for those sharing the same A. The only difference between this query and the example 6.1 in [6] is that a `Max` aggregate in involved in the nested correlated right-hand side instead of `Sum`.

Since equality is used in the predicate, according to the idea in [6], we prefer to use the index Partial Aggregate Indexes (PAI). With the extension covered in section 4.2.5, to handle the deletions for `Max`, another binary search tree (BST) of the data will be used for each different A value rather than only keeping the aggregate value. That is because the updated aggregate value cannot be recovered for deletions of `Max` operation.

6.1.1 Using PAI Index for Query with Sum Predicate. In the figure 4a, we show the code generated by [6] for example 6.1 in Python. Next, we will recap the steps for the incrementalization. Whenever a new tuple t comes, the `lhs_sum` in the sub-query will update by $t.X * t.B$. However, the `lhs_sum` is the same for every outer tuples. Thus, the PAI uses the `map2` to maintain it in $O(1)$ (Line 14). Secondly, the `rhs_sum` for the $t.A$ also changes by $t.X \times t.B$, the PAI maintains the `rhs_sum` by `map3` for each different A value in constant time (Line 13). Finally, RPAI utilizes the `map3` to keep the final aggregate answer ($sum(r.A * r.B)$) for all the `rhs_sum`. Since it is a hashmap, so all the updates (Line 16-18) can be done in $O(1)$. To return the updated final answer, we just need to probe the new `lhs_sum` into the `aggrMap` and then return the value.

6.1.2 Using PAI Index + BST for Query with Max Predicate. To illustrate the way to enhance PAI by another binary search tree to support deletions for `Max`, we convert the code in figure4a to the code in figure 4b. As you can see in the Line 4, the original `map3` has been improved from a (int -> int) hashmap to a (int -> binary search tree) hashmap. To find the `oldMaxB`(line 8) and the `newMaxB`(line 17), we can define a `getMax` method for the binary search tree. For insertion ($t.X = 1$) or deletion ($t.X = -1$) at line 12, we can also

```
 1 # primary maps
 2 map1 = {}    # A -> sum(A * B)
 3 map2 = 0.0   #   -> sum(B)  [lhs_sum]
 4 map3 = {}    # A -> sum(B)  [rhs_sum]
 5 # aggregate maps
 6 aggrMap = {} # rhs_sum -> sum(A * B)
 7
 8 def on_new_R(t: R):
 9   oldSumB = map3[t.A] # old rhs_sum for t.A
10   oldFinalAggSum = map1[t.A]
11
12   # update the maps
13   map3[t.A] += t.B * t.X
14   map2 += t.B * t.X
15   map1[t.A] += t.A * t.B * t.X
16   aggrMap[oldSumB] -= oldFinalAggSum
17   aggrMap[oldSumB + t.X * t.B]
18   += oldFinalAggSum + t.A * t.B * t.X
19
20   #compute the final result
21   res = aggrMap[map2 * 0.5]
22   return res
```

$\boxed{O(1)}$

(a) PAI for Sum-Equality based Nested Query (Section 6.1.1)

```
 1 # materialized views & aggregate maps
 2 map1 = {}  # A -> sum(A * B)
 3 map2 = 0.0 #   -> sum(B)
 4 map3 = {}  # A -> max(B) // BST for max(B)
 5 aggrMap = {} # rhs_max -> sum(A * B)
 6
 7 def on_new_R(t: R):
 8   oldMaxB = map3[t.A].getMax()
 9   oldFinalAggSum = map1[t.A]
10
11   # update the maps
12   map3[t.A].insert(t.B, t.X)
13   map2 += t.B * t.X
14   map1[t.A] += t.A * t.B * t.X
15   aggrMap[oldMaxB] -= oldFinalAggSum
16
17   newMaxB = map3[t.A].getMax();
18   aggrMap[newMaxB] += oldFinalAggSum + t.A * t.B * t.X
19
20   #compute the final result
21   res = aggrMap[map2 * 0.5]
22   return res
```

$\boxed{O(\log(|R|))}$

(b) PAI with BST for MAX-Equality based Nested Query (Section 6.1.2)

Figure 4: Code corresponding to PAI index with BST for the query in Example 6.1 and 6.2 (assuming $O(1)$ hash map access).

define a `insert` for the binary search tree. To get the best time complexity, we can use the Red-Black Trees [15], Left-leaning Red-Black Trees [32], Splay [33] or other balanced binary search tree supporting logarithmic insertions or deletions. Therefore, the time complexity for `insert` (line 12) is $\log(|R|)$, the time complexity for `getMax` is $\log(|R|)$. The total time complexity for the update is also $\log(|R|)$.

So far, we have seen that the additional balanced binary search tree works well which can update all the materialized views and aggregate maps in $O(\log(n))$ time. The logarithmic term can hardly be removed since the lower bound for maintaining the order of a sequence of data with size n is $O(\log(n))$.

6.2 Nested Max with Inequality

Example 6.3. *This query is from the example 2.2 in [6], it is the volume-weighted average price (VWAP, a technical analysis indicator used in finance).*

```
SELECT Sum(b.price * b.volume) FROM bids b
WHERE
0.75 * (SELECT Sum(b1.volume) FROM bids b1)
< (SELECT Sum(b2.volume) FROM bids b2
WHERE b2.price <= b.price)
```

Correspondingly, we create another example for the `Max`-based sub-query version of the example 6.3.

Example 6.4. *This query applies the* `Max` *in the predicate of the sub-query as the example 6.3*

```
SELECT Sum(b.price * b.volume) FROM bids b
WHERE
0.75 * (SELECT Sum(b1.volume) FROM bids b1)
< (SELECT Max(b2.volume) FROM bids b2
WHERE b2.price <= b.price)
```

Similarly, let us review the RPAI index proposed in [6] which is invented for such nested correlated inequality query as example 6.4.

6.2.1 Using RPAI Index for Query with Sum Predicate. In the figure 5a, the code generated by [6] for example 6.3 in Python in shown. When a new tuple t is delivered, since the `lhs_sum` does not change, so we can reuse the previous `map2` to maintain it. As for the `rhs_sum`, it computes the summation of `volume` for those records whose `price` are less than or equal to the `price` of the current record. We can use the RPAI index to maintain between `price` and `rhs_sum` in $O(\log(|R|))$ (mainly by efficient `getSum`). Therefore, we just need $O(\log(|R|))$ time to get the old `rhs_sum` (line 7) and the sum of all `volume` for `t.price` (line 8). Next, updating the `map3` can also be done in $O(\log(|R|))$ and updating the `map2` in constant time. Furthermore, since `rhs_sum` involves in all the records whose *price* are less than or equal to the *price* of the current record, a new record t will also influence the `rhs_sum` for those records whose *price* are greater than or equal to the *t.price*. Thus, when insert record t, we need to update a range of `rhs_sum` of records. Due to the logarithmic `shiftKeys` of RPAI, we can finish it very fast (line 10). So does update for `aggrIndex` (line 14-15). As for the final results, we can query the `getSum` by the `lhs_sum` to get it.

The key of the `aggrIndex` above is the `rhs_sum`. RPAI uses a variant of the binary search tree to maintain it. The main contribution of the RPAI is its clever design for the semantics of the node, which is the key relative to its parent. So that we can shift keys of a sub-tree by just shifting (adding or subtracting some value to) the root of that sub-tree.

6.2.2 Using RPAI Index for Query with Max Predicate. So far, we have seen the power of RPAI. As discussed in the limitation of [6], the RPAI does not work for `Max`/`Min`. The reason is that `Max`/`Min`

```
1  aggrIndex = {}  # <rhs_sum> --> sum(price * volume)
2  map2 = 0.0  # sum(volume)
3  map3 = {} # price --> sum(volume)
4
5  def on_new_bids(t: record):
6    # rhs_sum for new record (before update)
7    rhs_sum = getSum(map3, t.price)
8    volume = map3[t.price]
9    # update the aggregate index
10   shiftKeys(aggrIndex, rhs_sum-volume, t.X * t.volume)
11   # update the maps
12   map3.insert(t.price, t.volume*t.X)
13   map2 += t.volume*t.X
14   aggrIndex[rhs_sum + t.X * t.volume]
15   += t.X * t.price * t.volume
16   #compute the output
17   return compute()
18
19 def compute():
20   lhs_sum = map2 * 0.75
21   res = getSum(aggrIndex, inf)
22   - getSum(aggrIndex, lhs_sum)
23   return res
```

$\boxed{O(\log(|R|))}$

(a) RPAI for Sum-Inequality based Nested Query (Section 6.2.1)

```
1  maxIndex = {} # price --> rhs_max
2  sumIndex = {} # price --> sum(price * volume)
3  map2 = 0.0  # sum(volume)
4
5  def on_new_bids(t: record):
6    # update the lhs_sum for new record
7    map2 += t.volume*t.X
8    # insert new record
9    maxIndex.insert(t.price, t.X, t.volume)
10   sumIndex.insert(t.price, t.X*t.price*t.volume)
11   # update the rhs_max for maxIndex
12   updateMax(maxIndex, t.price, inf, t.volume, t.X)
13   # update the sum for merge index
14   updateSum(sumIndex, t.price, inf, t.X*t.price*t.volume)
15
16   # compute the output
17   return compute()
18
19 def compute():
20   lhs_sum = map2 * 0.75
21   rhs_max = getMax(maxIndex, lhs_sum)
22   res = getSum(sumIndex, inf) - getSum(sumIndex, rhs_max)
23   return res
```

$\boxed{O(\log(|R|))}$

(b) RPAI with BST for Max-Inequality based Nested Query (Section 6.2.2)

Figure 5: Code corresponding to PAI index with BST for the query in Example 6.1 and 6.2 (assuming $O(1)$ hash map access).

is not streamable. We will investigate more thoroughly the reason why `Max/Min` can not be supported. In figure 5a, we have seen the code generated for Sum-Inequality based nested query. Let us follow the same idea of using BST on the RPAI, convert the value of `map3` (line 3) from tracking the `Sum` to the `Max` for each key. Secondly, the key of `aggrIndex` (line 1) will be transformed to `rhs_max`.

Next, we will explore whether `getMax` (respective to `getSum`), `shiftKeys`, and `insert` can perform smoothly. Let us consider the `map3` at first, `insert` and `getMax` is related to `map3`. For `getMax`, we will follow the same idea to get the maximum `volume` among those whose `price` is no greater than the `price` since `Sum` and `Max` are both decomposable. When it comes to the `insert`, for the same key (`price`), we will keep the maximum `volume`. Same as the case in figure 4b, when we try to

If we try to convert sum-based RPAI to a max-based RPAI, it will be hard to define the semantics of the keys. Let us consider the example 5b, if we follow the same idea of RPAI, let us first use the `rhs_max` (re-define it since we are using `Max` instead of `Sum`) as the key of the `aggrIndex`. Then what is one node's key relative to its parent?

There are two cases, right child and left child. If you build a RPAI for `Max`-based query, assume r is the root of this tree, c_1 and c_2 are the left child and right child of r. Based on the definition of `rhs_max`, the key value of r is the maximum *volume* for those whose *price* is less than or equal to the *price* of the node r (including r and its left sub-tree). Since the `rhs_max` of c_2 is the maximum *volume* for r, its left sub-tree, c_2 and its left sub-tree. Thus, the difference between them are the maximum *volume* of c_2 and its sub-tree. If S and T are two sets, then the following equation holds:

$$Max(S \cup T) = Max(Max(S), Max(T)).$$

Similar to the original RPAI, we can define the key value of c_2 as the maximum *volume* of c_2 and its left sub-tree.

When it comes to the c_1, the semantics becomes much more vague. Since the `rhs_max` of c_1 only covers the maximum *volume* of itself and its left sub-tree. Consequently, the difference between c_1 and r will be r itself and r's right sub-tree. Is remains unclear how to define minus when the `Max` operation applied on sets.

On the other hand, we can try to build auxiliary BST as what we did for Query with equality. Following this way, we can remove the parent-relative keys and resume the `rhs_sum` semantics of each key. Thus, the update on one node will not influence the whole sub-tree, the high-performance sub-tree `shiftKeys` will fall back to $O(|R|)$ from $O(\log(|R|))$.

In essence, we can map the addition of `Sum` to the set union of `Max`, but we can hardly find the correspondence for the subtraction of `Sum` in set operations.

6.2.3 Using Range Index for Query with Max Predicate. As you observe from the Python code from figure 5a, we build `map3` from *price* to the `rhs_sum` and we also build `aggrIndex` from `rhs_sum` to the final aggregate result. `rhs_sum` grows monotonically with the increasing order of *price*. The final aggregate result increases similarly with the `rhs_sum`, so it also grows monotonically respective to the increasing order of *price*. Therefore, rather than build index based on `rhs_sum` (`aggrIndex`), a term hard to maintain, why not build index just based on *price* which is simple, without aggregation and keep the same monotonicity?

As shown in figure 5b, we build only two indices `maxIndex` (line 1) whose key and value are the *price* and `rhs_max` respectively, and `sumIndex` (line 2) whose key and value are the *price* and the final aggregate result. `maxIndex` is similar to the `map3` (line 3 of figure 5a), but directly keeps the prefix aggregation of *volume* (range value) on each node rather than only maintaining *volume* for specific *price* (just like `map3`). For `map3`, query the `rhs_sum`, we need to aggregate all the sub-trees' value who are smaller than the current *price*. Right now, the value on one specific node is enough. For simplicity, you can regard the `sumIndex` and `maxIndex` as two binary search trees who is capable of range update.

When a new record t comes, we will insert or delete (depends on $t.X$) the new record (line 9-10) firstly. Next, we will fill the `rhs_max` and final aggregate result for it. `updateMax` will be used to update the `rhs_max` for those whose *price* is between $t.price$ and inf by inserting or deleting one $t.volume$. As for the `updateSum`, we can do something similar for the `sumIndex` by adding $t.X * t.price * t.volume$. Concretely, `updateSum` and `updateMax` are range update in BST, we can use a tag on the root to represent the update for the whole sub-tree. When we need to dive into this sub-tree (query or update), we can push down the tag, we call such operation as "lazy update". It only costs $O(\log(|R|))$ time. All the details about these range update will be covered in the section **??**. All the operations related to `map2` does not change (line 2, 6). When computing the final results, we can utilize the `getSumByMax` function which is used for computing `Sum` of the second value by probing the `rhs_max`. Since we know both values are growing monotonically in the ascending order of *price*. It also costs $O(\log(|R|))$ time. Therefore, the total time complexity is still $O(\log(|R|))$.

7 BINARY SEARCH TREE WITH RANGE UPDATES

In this section, we design a new tree-based data structures called Range Binary Search Tree (RBST) to support both range query for `Max` and `Sum` in logarithmic time. As a augmented binary search tree, it also supports common tree operations (e.g. `insert`, `delete`).

Recall that the problem we are facing is when constructing the parent-relative keys for `Max`-based query, there is no way to build parent-relative keys but we still want an efficient range update operation (`shiftKeys` can be regarded as a range updates for the keys). Based on the range search in [12], we know that a range query on a binary search tree can be done by merging all the results (e.g. `Sum`, `Max`) from different disjoint sub-trees since these aggregate operations are decomposable.

Follow this way, a range update can be decomposed into several disjoint sub-tree updates. Now that the only problem remains is how to do range update without parent-relative keys. Think about the example in figure 6. Same as the case in example 6.4, the node's value is the prefix maximum value in the tree in the ascending order of the key. The only one operation we can apply is updating a range of keys' value. For the operation `updateMax(index, 5, inf, 45, 1)`, our goal is to update the node whose key is larger than or equal to 5 by the new value 45.

To do the range update efficiently, we add a tag for each node to do lazy update. Following the same idea of `getSum` in [6], we can locate the node sharing the same key as the new record along a path from the root to one internal or leaf node. Along the way to search for the query range, we check every node's key and its sub-tree range. If the whole sub-tree is included in the update range (5-inf), then we will change the tag of root and update its value instantly (e.g. node with key 8). Otherwise, we check whether the node itself is in the query range, then we update its value directly (e.g. node

Key	2	3	5	6	7	8	9
Value	7	10	15	13	14	27	33

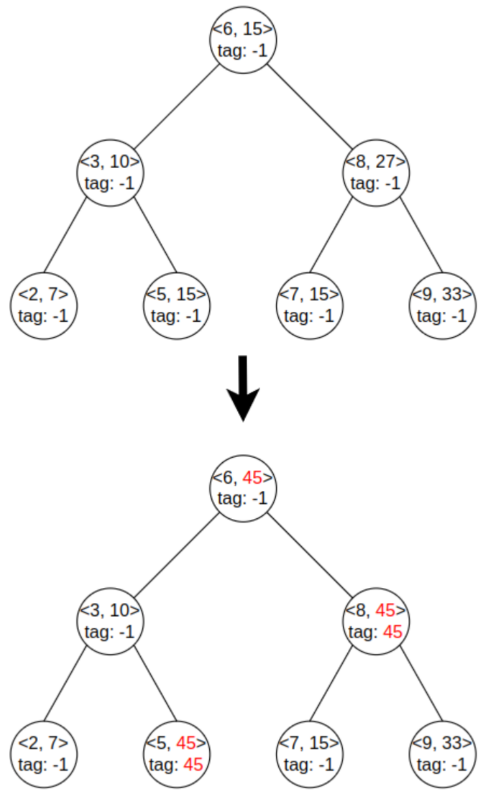


Figure 6: An example run of the `updateMax(index, 5, inf, 45, 1)` in a RBST. The original key-value pairs are `(2,7), (3,10), (5, 15), (6, 12), (7, 14), (8, 27), (9, 33)`. The value maintains the prefix maximum value in the increasing order of keys.

with key 5 & 6). Next, we need to decide whether to search into sub-trees of the current node. If its sub-tree's range is intersected with the update range, we will update further (e.g. search in the right sub-tree of node with key 3). If not, we will not search into its child (e.g. no search in the left sub-tree of node with key 3).

Following this idea, we can traverse all the node or sub-tree whose key range is intersected with the update range and update them. As you can image, if it is a `sumIndex`, the tag will represent the value added to all the node in some sub-tree. As for the query, we will do "lazy update" which is to push-down the tag into sub-trees along the way to search the query range. In figure 7, there is an example for push-down tag. The query range is just the node with key 7 and there is a tag on node with key 8. Since there is a tag on node with key 8, we need to push-down the tag to its sub-trees, otherwise, its sub-trees will lose the information of such tag which should be sent to them earlier. Thus, the push-down operation will update the value of the tag of its sub-trees. The push-down

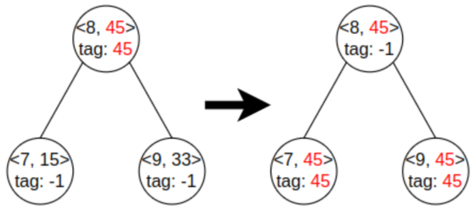


Figure 7: An example push-down in a RBST from node with key 8 to its sub-trees.

operation can be processed when any update or query is issued into some sub-tree. If there is no update or query in some sub-tree, we can let some its ancestor keep the information forever. Thus, the tag help us to make the range update and range query much more efficient without using the parent-relative keys.

As for the time complexity, a range update or query can be decomposed by $O(\log(n))$ sub-tree update or query where n is the total number of node. Additionally, push-down, tag and value update are both $O(1)$. Therefore, the total time complexity for RBST is $O(\log(n))$ and it can be used to support nested correlated query with `Max`/`Min` aggregation.

8 GENERAL IVM EVALUATION FOR JOIN

When it goes back to the plain query plan (by SparkSQL), we can find that the optimization techniques covered by RPAI and PAI [6] and section 6 using the join condition (both left and right sides) in the sub-query as the key and the aggregate value of the outer query as the value to build the index. Therefore, that is the special case for `Groupjoin`. Next, we will cover more general IVM evaluation strategy for `Join` oprtator.

We have covered the cases for equality involved (including anti-, semi-) joins in the end of secion 4. In DBToaster [21], the author has proposed that materializing all the intermediate results will be costly, then they choose to select several sub-view to materialize. If there is a multiple (more than 2) way join in one query, it is better to maintain indexes for each relation by query decomposition. Concretely, when there is au update from one relation involving in a 3-way join, the new inserted/deleted tuple will be probed to the indexes of the other two relations to update the final materilized view. In this way, no join results for any two relations are maintained, as you can image, it may occur $O(|R_i| \times |R_j|)$ join results where R_i and R_j are two relations.

Case will be very similar for the inequality involved join. Let us consider the join condition $R_1.A < R_2.B$ where R_1, R_2 are two relations, and A, B are their attributes. Instead of materializing all the join results, we only build two indexes to maintain the order for $R_1.A$ and $R_2.B$ separately. When the user wants to know the final answer, we just need to issue the join between them. As for the case when one side is aggregate sub-query, we only need to create and keep the index for another side, when the trigger for the final result is issued, we can easily probe the number of that sub-query to the index from the other side.

9 RELATED WORK

Materializing and reusing query results for other queries has been an area of exploration for quite some time. Approaches range from sub-query materialization [31] and common sub-expression extraction [43], to multi-query optimization [31]. Building on the concept of materializing additional first-order views [31], DBToaster [20, 21] employs the idea of IVM recursively, using derivatives to compute the original function. However, this approach is limited to cases where the sub-query is simpler than the outer query. In general, these studies focus on strategies for determining what to materialize in IVM systems.

Efficiently evaluating incremental views requires striking a balance between update workloads and query latency. Eager and lazy methods have been proposed in [10, 42] to address this trade-off. Additionally, incrementability is introduced by [36] as a measure of IVM's cost-effectiveness, helping to decide whether to apply eager or lazy execution to a query. In terms of the resource consumption, [35] studies how to efficiently evaluate queries under constraining resource, such as limited memory or disk space. In addition, [23, 37] attempt to share resources across different queries in data streams. Besides tuple-level IVM, [35, 38, 39] explore how to evaluate for intermittent data ingestion, that is to say data will come incrementally in a predictable style (e.g. once per hour). However, none of them builds a framework to optimize for IVM completely.

With numerous materialization strategies available, [20, 21, 39] study how to design a query compiler for IVM. DBToaster [20] proposes a query language *AGgregate CAlculus* (AGCA) for tuple. But it lacks ability to express order (e.g. `Sort, Min, Max`) in an efficient way. When it comes to nested queries, the code generated by DBToaster yields sub-optimal performance. Tempura [39] proposes a time-varying relation (TVR) [7] based Incremental query Planning (TIP for short) Model, which incorporates temporal information to allow the optimizer to explore efficient plans as ingested data changes. As a result, there are many opportunities for improving performance of planned views without considering future data ingestion. Recently, [6] proposes PAI and RPAI to optimize correlated nested aggregate queries for IVM. Besides, it introduces incrementalizing Algorithm to extract query patterns fit their advanced data structures. However, rewriting queries using their representation is complex and challenging to integrate into modern database systems.

[14] claims that coarser periodic refresh of IVM can be regarded as sliding window, which enhances the role of IVM in stream processing. Data stream processing systems [9, 41] are designated for window-based queries (e.g. tumbling windows, sliding windows). Naiad (Timely Dataflow, TD for short) [26], Differential Dataflow (DD for short) [24] are designed towards distributed stream data processing. Based on them, Materialize [17] builds a streaming database, focusing on high performance IVM. The fundamental data representation aligns with the bag relational algebra, which means everything involves multiplicities. Materialize explores some join optimizations, including delta computation and public indices for a single data source, which are similar to our backend operations. However, it has not yet solved aggregations like `Max`/`Min`, and operators that do not satisfy subtraction are unsupported. The same limitation applies to `Sort`. Moreover, Materialize must maintain a

large volume of intermediate results for multi-way joins. We believe that integrating our `Group-join` and `BST` could significantly improve their performance.

Some DBMSs (e.g. PostgreSQL [2]) also planned to support IVM gradually [27, 28]. Taking inspiration from ID-based [18], one OID-based approach was proposed to handle the view maintenance for `Join` operator. The OID concept involves materializing an ID map between join results and the two joined tables. Consequently, when an update occurs, the OID can be used to delete outdated results after computing the updated join results. Besides `Inner Join`, `Select`, and `Project` have also been supported. Surprisingly, `Distinct` is handled by multiplicities-based counting algorithm in Postgresql. However, it is not able to handle other aggregation operators and `Group-By`. Other data warehouse product (e.g. Amazon Redshift [1]) would like to support incremental refresh views but only limited in the scope of flat select-project-join-aggregate (SPJAG) queries [3] (e.g. without sub-query).

10 CONCLUSION

In summary, we have highlighted the benefits of using query plans for incremental view maintenance, such as ease of use, flexibility, and the ability to apply traditional query optimization techniques. Previous work has struggled to address certain aspects, such as `MAX/MIN`, order-unfriendly representations from DBToaster [21], and inefficient evaluation strategies in some commercial databases. In this report, we first outline how to structure the IVM evaluator and the IVM evaluation strategies for common operators in query plans. Next, we address the limitation of missing `MAX/MIN` for aggregate nested correlated queries by introducing a new index - the range binary search tree. We then demonstrate how to extend the optimization to more general `Join` operators. As a result, we present a framework that integrates query plans into an IVM compiler, complete with numerous specialized optimization strategies.

REFERENCES

[1] 2022. *Amazon Redshift: Data warehousing reinvented for an ever-changing data landscape.* https://aws.amazon.com/redshift/

[2] 2022. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* https://www.postgresql.org/

[3] 2022. *REFRESH MATERIALIZED VIEW - Amazon Redshift.* https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh-sql-command.html

[4] 2023. *Refreshing Materialized Views.* https://docs.oracle.com/database/121/DWHSG/refresh.htm#DWHSG-GUID-64068234-BDB0-4C12-AE70-75571046A586

[5] 2023. *TPC-H Homepage.* https://www.tpc.org/tpch/

[6] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). In *SIGMOD Conference*. ACM, 136–149. https://doi.org/10.1145/3514221.3517889

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[8] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[10] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for Deferred View Maintenance. In *SIGMOD Conference*. ACM Press, 469–480. https://doi.org/10.1145/233269.233364

[11] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. 2007. Execution strategies for SQL subqueries. In *SIGMOD Conference*. ACM, 993–1004. https://doi.org/10.1145/1247480.1247598

[12] David Eppstein. 2021. *Range search and augmented binary search trees.* https://www.ics.uci.edu/~eppstein/261/s21w7.pdf

[13] César A. Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD Conference*. ACM, 571–581. https://doi.org/10.1145/375663.375748

[14] Thanaa M. Ghanem, Ahmed K. Elmagarmid, Per-Åke Larson, and Walid G. Aref. 2010. Supporting views in data stream management systems. *ACM Trans. Database Syst.* 35, 1 (2010), 1:1–1:47. https://doi.org/10.1145/1670243.1670244

[15] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *FOCS*. IEEE Computer Society, 8–21.

[16] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD Conference*. ACM, 1259–1274. https://doi.org/10.1145/3035918.3064027

[17] Materialize Inc. 2021. *Materialize: Event Streaming Database for Real-Time Applications.* Retrieved June 22, 2021 from https://materialize.com/

[18] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. 2015. Utilizing IDs to Accelerate Incremental View Maintenance. In *SIGMOD Conference*. ACM, 1985–2000. https://doi.org/10.1145/2723372.2750546

[19] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *PODS*. ACM, 87–98. https://doi.org/10.1145/1807085.1807100

[20] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278.

[21] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. https://doi.org/10.1007/s00778-013-0348-4

[22] Per-Åke Larson and Jingren Zhou. 2007. Efficient Maintenance of Materialized Outer-Join Views. In *ICDE*. IEEE Computer Society, 56–65. https://doi.org/10.1109/ICDE.2007.367851

[23] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.* 13, 10 (2020), 1793–1806. https://doi.org/10.14778/3401960.3401974

[24] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[25] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *Proc. VLDB Endow.* 4, 11 (2011), 843–851. http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf

[26] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[27] Yugo Nagata. 2018. *Implementing Incremental View Maintenance on PostgreSQL.* https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2195/slides/144/Implementing%20Incremental%20View%20Maintenance%20on%20PostgreSQL%20.pdf

[28] Yugo Nagata. 2019. *Toward Implementing Incremental View Maintenance on PostgreSQL.* https://www.sraoss.co.jp/wp-content/uploads/files/event_seminar/material/2019/Toward-Implementing-Incremental-View-Maintenance.pdf

[29] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[30] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*. Morgan Kaufmann, 802–813.

[31] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD Conference*. ACM Press, 447–458. https://doi.org/10.1145/233269.233361

[32] Robert Sedgewick. 2008. *Left-Leaning Red-Black Trees.* https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf

[33] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686.

[34] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322. https://doi.org/10.1145/3183713.3196893

[35] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent Query Processing. *Proc. VLDB Endow.* 12, 11 (2019), 1427–1441. https://doi.org/10.14778/3342263.3342278

[36] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. Thrifty Query Execution via Incrementability. In *SIGMOD Conference*. ACM, 1241–1256. https://doi.org/10.1145/3318464.3389756

[37] Dixin Tang, Zechao Shang, William W. Ma, Aaron J. Elmore, and Sanjay Krishnan. 2021. Resource-efficient Shared Query Execution via Exploiting Time

Slackness. In *SIGMOD Conference*. ACM, 1797–1810. https://doi.org/10.1145/3448016.3457282

[38] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing. In *SIGMOD Conference*. ACM, 2797–2800. https://doi.org/10.1145/3318464.3384708

[39] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing. *Proc. VLDB Endow.* 14, 1 (2020), 14–27. https://doi.org/10.14778/3421424.3421427

[40] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDB J.* 12, 3 (2003), 262–283. https://doi.org/10.1007/s00778-003-0107-z

[41] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*. ACM, 423–438. https://doi.org/10.1145/2517349.2522737

[42] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. 2007. Lazy Maintenance of Materialized Views. In *VLDB*. ACM, 231–242. http://www.vldb.org/conf/2007/papers/research/p231-zhou.pdf

[43] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD Conference*. ACM, 533–544. https://doi.org/10.1145/1247480.1247540