Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI)

Supun Abeysinghe, Qiyang He, Tiark Rompf
{tabeysin,he615,tiark}@purdue.edu
Purdue University
West Lafayette, Indiana

Abstract

Incrementalization of queries is imperative in cases where data arrives as streams and output is latency-critical and/or desired before the full data has been received. Incremental execution computes the output at a given time by reusing the previously computed outputs or maintained views rather than re-evaluating the query from scratch. There are various approaches to perform this incrementalization ranging from query-specific algorithms and data structures (e.g., DYN, AJU) to general systems (e.g., DBToaster, Materialize).

DBToaster is a state-of-the-art system that comes with an appealing theoretical background based on the idea of applying Incremental View Maintenance (IVM) recursively, maintaining a hierarchy of materialized views via delta queries. However, one key limitation of this approach is its inability to efficiently incrementalize correlated nested-aggregate queries due to an inefficient delta rule for such queries. Moreover, none of the other specialized approaches have shown efficient ways to optimize such queries either. Nonetheless, these types of queries can be found in many real-world application domains (e.g., finance), for which efficient incrementalization remains a crucial open problem. In this work, we propose an approach to incrementalize such queries based on a novel tree-based index structure called Relative Partial Aggregate Indexes (RPAI). Our approach is asymptotically faster than other systems and shows up to 1100× speedups in workloads of practical importance.

CCS Concepts

• **Theory of computation → Data structures and algorithms for data management**; • **Information systems → Query optimization**.

Keywords

incremental processing; nested aggregate queries; aggregate indexes; parent-relative trees

ACM Reference Format:
Supun Abeysinghe, Qiyang He, Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517889

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517889

1 Introduction

Many real-world systems perform analytical queries on continuously arriving streams of data that result in dynamic datasets with high update rates. These types of workloads are prevalent in many modern big data application domains including finance, social media, etc., and generally focus on extracting insights, trends, and anomalies from data streams [22]. For example, in algorithmic trading, such queries are performed on fast arriving streams of data to compute key metrics that drive trading decisions [22].

This is generally done in the form of incremental processing where, given a query Q, a database db, the task is to efficiently compute $Q(db + \Delta db)$ under updates Δdb using $Q(db)$ (i.e., the previous result) and any additionally maintained auxiliary data structures. Incremental processing is not just useful for processing streams of data, but also in other use cases such as, producing low-latency approximate results for batch queries by operating on mini-batches of data [46], and eager processing of queries (i.e., before all the data is ready) to produce faster results with limited resources [40], intermittent query processing [39], etc.

Traditional relational databases (DBMS) perform this in the form of Incremental View Maintenance (IVM) when maintaining materialized views. Specifically, users can materialize query outputs (e.g., for faster retrieval) and the DBMS engine maintains these views under updates to the underlying data sources by incrementally updating the view accordingly (via *delta queries*). This is generally faster compared to evaluating the query from scratch on the updated database. Although these traditional IVM techniques handle simple queries well, they fall back to re-computation for complex queries with nested and correlated subqueries [24].

DBToaster [22, 24] is a state-of-the-art incremental query evaluation system that takes this notion of delta queries and applies it recursively (i.e., delta of delta queries and so on) so that the views are maintained by not just a single materialized structure, but a set of auxiliary views corresponding to each level of the recursive delta query. This notion of higher-order delta queries works well in practice and has shown significant performance gains over commercial DBMS and Stream Processing systems [25].

However, this recursive approach is only effective for a class of queries that is guaranteed to have delta queries simpler than the higher-level query (further discussed in Section 2.1.2). Importantly, queries with nested aggregates do not satisfy this key property. Specifically, the delta of such queries is simply running the query twice and computing $Q(db+\Delta db) - Q(db)$ which is worse than simply re-evaluating the query [29]. Therefore, systems like DBToaster fall back to re-computing for such queries. In Section 2.2, we show that DBToaster takes $O(n^2)$ time to maintain such a query under updates which is the same asymptotic time as re-computation whereas

we demonstrate the same can be done in $O(\log n)$ time by using an additional index to maintain partial aggregates.

Follow-up work on DBToaster [29] presented a technique called *domain extraction* for specifically optimizing this re-evaluation strategy by narrowing down the iteration space. However, this only works for cases where the nested aggregates are correlated to the outer query on equality predicates. Even for the queries that use this technique, we show in Section 2.1.2 that this approach yields sub-optimal asymptotic performance (e.g., takes $O(n)$ where $O(1)$ is possible). Another line of related work focuses on building specialized data structures and algorithms to efficiently incrementalize certain classes of queries (e.g., acyclic conjunctive queries (CQ) with equalities [16] or inequalities [17], acyclic foreign key joins [42] etc.). However, the presented specialized data structures either do not support or do not efficiently handle nested aggregate queries.

In this work, we focus on improving the incrementalization efficiency of aggregate queries containing nested aggregate subqueries in their predicates. First, we introduce a general algorithm that is based on the idea of identifying and updating only the aggregate values affected by a tuple insertion (or deletion). This is done by analyzing the correlated columns of inner queries and the corresponding uncorrelated columns used in the inner query predicates. Second, we focus on a specific subset of queries that appear commonly in practice (especially in finance-related use cases) and propose a way to further optimize them. Specifically, we build additional index structures that are indexed by aggregate values such that a range of aggregate values can be shifted efficiently, providing the ability to compute the final result directly from these indexes. We observe that none of the existing data structures support the required operations in a reasonable time, and hence, design a novel tree-based data structure called relative partial aggregate index (RPAI) that stores aggregate values (i.e., keys) in a parent-relative manner to enable range key shifting in logarithmic time. Our analysis shows that the use of these aggregate indexes result in significant asymptotic speedups (e.g., $O(n^2)$ to $O(\log n)$) and up to 1100× speedups in workloads of practical importance.

This paper makes the following specific contributions.

- We present a case study with two examples of nested aggregate queries and analyze how existing approaches handle those queries. Then, we motivate our approach of using aggregate indexes (PAI Maps and RPAI Trees) and demonstrate how such structures can improve the incrementalization (Section 2).
- We design an efficient tree-based data structure for RPAI and present algorithms for the key operators with an analysis of their time complexity (Section 3).
- We present a novel general algorithm for incrementalizing correlated nested aggregate queries, followed by further optimizations using PAI Maps and RPAI Trees. Then, we discuss the limitations and overheads associated with our approach. (Section 4).
- We evaluate the performance of our algorithm against DBToaster, a state-of-the-art system that supports incremental execution of SQL queries, and show that our approach performs significantly better in real-world datasets in line with the expected performance behaviors due to asymptotic speedups (Section 5).

We discuss related work in Section 6 and present concluding remarks in Section 7.

2 Incrementalization of Nested Aggregates

In this section, we consider two example nested aggregate queries of different forms and explore how existing systems and approaches handle these queries. Then, we identify the key limitations of these approaches, which motives the ideas presented in this work. Specifically, we demonstrate how our approach can improve the efficiency of incrementalization of these queries.

2.1 Nested Aggregate with Equality

Example 2.1. *Consider the following simple nested aggregate query that only consists of equality predicates. This computes an aggregate of values that is responsible for a given fraction ($\frac{1}{2}$ in this case) of all the records. Queries of this structure are commonly found in many use cases [22, 38].*

```
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 query operates on a single relation $R(A, B)$ and consists of two nested aggregate queries. The inner query on the left-hand side is not correlated with the outer query whereas the right-hand sub-query is correlated implying that the corresponding aggregate value varies for different records from the outer query.

2.1.1 Naive Re-evaluation Strategy Figure 1a shows a naive re-evaluation computation of the given query in Python. The code is self-explanatory and follows the same structure as the query (i.e., nested loops for the subqueries). Notice that this computation happens every time $R(A, B)$ gets updated (i.e., tuple insertions, or deletions). That is, when updates to R is ΔR, $Q(R + \Delta R)$ is evaluated from scratch irrespective of the fact that $Q(R)$ was previously computed. Due to the presence of nested loops, overall asymptotic time complexity of producing the final result upon updates to R comes down to $O(|R|^2)$ where $|R|$ is the number of tuples in R.

2.1.2 DBToaster Incremental Query Execution As discussed in Section 1, DBToaster falls back to recomputation due to the lack of an efficient delta query for queries containing aggregate subqueries in the predicates. Figure 1b shows the code generated by DBToaster for the query in Example 2.1. We converted the generated C++ code to Python to improve readability and combined the tuple addition and deletion triggers into a single (equivalent) trigger to make the code more concise ($t.X = 1$ for insertions and $t.X = -1$ for deletions).

The code shows that even though it relies on re-evaluation for connecting the nested aggregate with the outer query, other parts of the query are efficiently incrementalized (compared to naively re-evaluating). For instance, the `lhs_sum` computation which required iterating through all the records (line 8-11 in Figure 1a), is now computed in constant time by incrementally maintaining the sum of B values using `map2`. Since `rhs_sum` depends on the $r.A$ values from the outer query, `map3` maintains the relevant $sum(B)$ values for different $r.A$ values. Here, rather than re-evaluating all the `rhs_sum` values (lines 13-16 in Figure 1a), only the sum affected by the new tuple is updated. `map1` maintains the $sum(A * B)$ for each A value and is used to compute the final aggregate sum. Specifically, once `lhs_sum` and `rhs_sum` are computed, the set of A values that satisfy the condition are found and the corresponding $sum(A * B)$ values are summed up. Lines 9-11 in Figure 1b correspond to updating the maps based on the incoming tuple and lines 16-20 show the

```
R = []
def on_new_R(t: R):
  R.append(t)
  # re-evaluate from scratch
  res = 0.0
  for r in R:
    # evaluate lhs nested aggregate
      (uncorrelated)
    lhs_sum = 0.0
    for r1 in R:
      lhs_sum += r1.B
    lhs_sum *= 0.5
    # evaluate rhs nested aggregate
      (correlatead)
    rhs_sum = 0.0
    for r2 in R:
      if r2.A = r.A:
        rhs_sum += r2.B
    # evaluate the predicate and
     update aggregate
    if (lhs_sum == rhs_sum):
      res += r.A * r.B
```

$O(|R|^2)$

(a) Naive Re-evaluation (Section 2.1.1)

```
# materialized views
map1 = {}  # A -> sum(A * B)
map2 = 0.0 #   -> sum(B)
map3 = {}  # A -> sum(B)

def on_new_R(t: R):
  res = 0.0
  # update the maps
  map1[t.A] += t.A * t.B * t.X
  map2 += t.B * t.X
  map3[t.A] += t.B * t.X

  # compute lhs_sum
  lhs_sum = 0.5 * map2
  # find each rhs_sum
  for a in map1:
    rhs_sum = map3(a)
    # evaluate the predicate and
     update aggregate
    if lhs_sum == rhs_sum:
      res += map1(a)
  return res
```

$O(|R|)$

(b) DBToaster (partially incremental; Section 2.1.2)

```
# primary maps
map1 = {}    # A -> sum(A * B)
map2 = 0.0   #   -> sum(B)  [lhs_sum]
map3 = {}    # A -> sum(B)  [rhs_sum]
# aggregate maps
aggrMap = {} # rhs_sum -> sum(A * B)

def on_new_R(t: R):
  oldSumB = map3[t.A] # old rhs_sum for t.A
  oldFinalAggSum = map1[t.A]

  # update the maps
  map3[t.A] += t.B * t.X
  map2 += t.B * t.X
  map1[t.A] += t.A * t.B * t.X
  aggrMap[oldSumB] -= oldFinalAggSum
  aggrMap[oldSumB + t.X * t.B]
    += oldFinalAggSum + t.A * t.B * t.X

  #compute the final result
  res = aggrMap[map2 * 0.5]
  return res
```

$O(1)$

(c) Our approach based on aggregate indexes (fully incremental; Section 2.1.3)

Figure 1: Code corresponding to different execution strategies for the query in Example 2.1 (assuming $O(1)$ hash map access).

computation of the final result. The overall time complexity of maintaining the final result under updates to R is $O(|R|)$.

2.1.3 Using Aggregate Indexes A key observation at this point is that even though `lhs_sum` changes with incoming tuples, the aggregate value remains a fixed value for all the outer tuples as `lhs_sum` is not correlated to the outer query. Therefore, if we have an additional index that maps different `rhs_sum` values to the corresponding final aggregate sums (in this case `SUM(A * B)`), we can query that map using the current (updated) `lhs_sum` to compute the updated final result in constant time. However, populating such an aggregate index naively by computing `rhs_sum` for each tuple (using `map3`) would still leave the overall time complexity at $O(|R|)$.

Now the key question is, rather than populating this aggregate index by iterating and computing `rhs_sum` for all the tuples in R, is it possible to maintain it in less than $O(|R|)$ time. Observe that whenever a new tuple t arrives, three things change. First, the `lhs_sum` gets incremented by $t.B * t.X$ which can be updated in constant time. Second, the new record has a corresponding `rhs_sum` and that needs to be added to the `aggrMap`. This can also be done in constant time by computing the `rhs_sum` using `map3` as we saw before and updating the corresponding entry in `aggrMap`.

Third, the addition of the new tuple to $r2$ (in the right subquery) will trigger a change of `rhs_sum` values of all tuples having the same A value as $t.A$ (because the condition of the nested query predicate is $r.A = r2.A$). Therefore, all the tuples with the same A value will have the same `rhs_sum` because the set of tuples responsible for their respective `rhs_sum` are only the ones having the same A. That is, with the addition of t, the `rhs_sum` corresponding to $t.A$ gets incremented by $t.B * t.X$ and we need to update the `aggrMap` to reflect this change. We can do this by moving the corresponding `sum(A*B)` from the old `rhs_sum` key to `rhs_sum`+$t.B*t.X$. However, we cannot simply move the value for `rhs_sum` to `rhs_sum`+$t.B*t.X$ because there can be the same `rhs_sum` for different A values (i.e., different groups of tuples can have the same aggregate value). We can handle this by using `map3` to move the respective portion of the value from `aggrMap[old_rhs_sum]` (i.e., only $sum(A * B)$ of records having $t.A$ in A) to `aggrMap[new_rhs_sum]`.

We call indexes of type `aggrMap`, Partial Aggregate Indexes (PAI) since they store aggregates values as keys and map to aggregates. Figure 1c shows the implementation based on the PAI Map based approach. The entire update routine only contains updates to hash maps and does not require any form of iteration. Therefore, the query in Example 2.1 can be incrementally maintained in $O(1)$ time using our approach which is more efficient than re-evaluation which takes $O(|db|^2)$ time, and DBToaster which takes $O(|db|)$.

2.2 Nested Aggregate with Inequalities

In the previous example, we demonstrated how a query that has an $O(|R|^2)$ re-evaluating cost and an $O(|R|)$ incremental maintenance cost (in DBToaster) can be incrementalized in $O(1)$ time by using PAI Maps. The query consisted of just equality predicates, making it possible to use hash maps to maintain the aggregate indexes under updates in constant time. In this section, we consider a real-world query of a similar structure having inequality predicates.

Example 2.2. *The query computes the volume-weighted average price (VWAP) over bids that is in the final quartile (i.e, more than 75%) of total stock volume [22].*

```
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)
```

Example 2.2 is a query from the finance benchmark which is used in multiple other related works [22, 24, 25] to evaluate the performance of incrementalization. The query operates on traces

```
1  bids = []
2  def on_new_bids(t: record):
3    # update the base table
4    bids.addRecord(t)
5    # re-evaluate from scratch
6    res = 0.0
7    for b in bids:
8      # evaluate the lhs nested
         aggregate (uncorrelated)
9      lhs_sum = 0.0
10     for b1 in bids:
11       lhs_sum +=
12             b1.volume * b1.X
13     lhs_sum *= 0.75
14
15     # evaluate the rhs nested
         aggregate (correlated)
16     rhs_sum = 0.0
17     for b2 in bids:
18       if b2.price <= b.price:
19         rhs_sum +=
20               b2.volume * b2.X
21
22     # evaluate the predicate and
         update the aggregate
23     if (lhs_sum < rhs_sum):
24       res +=
25         b.price * b.volume * b.X
```

$O(|bids|^2)$

(a) Naive Re-evaluation (Section 2.2.1)

```
1  map1 = {}
2    # price -> sum(price*volume)
3  map2 = 0.0
4    # sum(volume)
5  map3 = {}
6    # price -> sum(volume)
7
8  def on_new_bids(t: record):
9    # update the maps
10   map1[a.price] += t.X * t.price
11                    * t.volume
12   map2 += t.X * t.volume
13   map3[a.price] += t.X
14                    * t.volume
15   #  evaluating correlated nested
        aggregate for each outer
        distinct price
16   res = 0.0
17   for b_price in map1:
18     rhs_sum = 0.0
19     for b2_price in map1:
20       if b2_price <= b_price:
21         rhs_sum += map3[b2_price]
22     # evaluating condition
23     if 0.75 * map2 < rhs_sum:
24       res += map1[t.price]
25   return res
```

$O(|bids|^2)$

(b) DBToaster (partially incremental; Section 2.1.2)

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

PAI Map	RPAI Tree
$O(\|bids\|)$	$O(\log \|bids\|)$

(c) Our approach based on aggregate indexes (fully incremental; Section 2.1.3)

Figure 2: Code corresponding to different execution strategies for the query in Example 2.2.

of records from order books that contain information about bids and asks for shares (or any other asset) in a financial market. These trades occur at rapid rates making it extremely important to have fast refresh rates as the queries compute key metrics that drive efficient algorithmic trading [22]. Moreover, the transactions in these financial markets often contain updates or retractions of older transactions, requiring the incremental query engines to maintain these queries incrementally under both insertions and deletions.

The *bids* relation consists of five attributes *timestamp, id, broker_id, volume, price*, plus an additional attribute (*bids.X*) to distinguish between record deletion (-1) and insertion (+1). Similar to Example 2.1, this query contains two nested aggregate queries with one of them being correlated to the outer query. The main difference is the use of inequality predicates in the query.

2.2.1 Naive Re-evaluation Strategy Figure 2a shows the code for a naive re-evaluation approach to compute the output of this query. Here, the *bids* is updated as records arrive and the result is computed from scratch by looping through the records while computing `lhs_sum` and `rhs_sum` using nested loops. The overall asymptotic time complexity is $O(|bids|^2)$ where $|bids|$ is the cardinality of the *bids* relation at the given point.

2.2.2 DBToaster Incremental Query Execution Figure 2b shows the code generated by DBToaster for the Example 2.2 query. DBToaster creates a set of maps representing intermediate materialized views to incrementally maintain the query. The two nested subqueries are fully incrementalized using two maps `map2` (for `lhs_sum`) and `map3` (for `rhs_sum`). `map1` maintains the sum of `price * volume` per each `price`. This is useful when computing the final result as we need to accumulate `price * volume` based on a set of `price` that satisfies the given conditions.

Similar to the previous example, DBToaster fails to incrementalize the computation of the final result that requires finding records from the outer query that satisfy the query conditions. Therefore, it falls back to computing the final result by iterating through records which is essentially similar to the naive re-evaluation strategy in Figure 2a (Lines 16-20). Hence, the asymptotic time complexity remains at the same level as naive re-evaluation, at $O(|bids|^2)$.

2.2.3 Using Aggregate Indexes to Fully Incrementalize To further incrementalize this query, we can follow a similar intuition to the previous example where we introduced the use of PAI Maps to index the final aggregate sum using `rhs_sum` as a key. Assuming such an index can be efficiently maintained, finding the final aggregate can be done in linear time by iterating through `rhs_sum` keys that are greater than `lhs_sum` and accumulating the corresponding `sum(price * volume)` values. For that purpose, we can define a `getSum(key)` method for hash maps that finds the sum of values having keys less than or equal to a given key by simply iterating over the keys. The `compute()` method in Figure 2c (lines 23-27) shows how to use `getSum` to compute the final result.

Now the question is: can we efficiently maintain the `aggrIndex` under updates? We cannot simply follow the same approach as before to shift the aggregates as the predicate inside the correlated nested subquery (i.e., `b2.price <= b.price`) is not an equality. Specifically, insertion (or deletion) of a new record t does not only

increment (or decrement) the rhs_sum of outer records having the same price as t.price but also the ones with price > t.price. This update cannot be done by simply iterating over the price values since the aggrIndex does not store the corresponding price values. For this, we can potentially use an additional index that maps price to the corresponding rhs_sum value (similar to map1 in Figure 2b).

Alternatively, rather than relying on an additional index, we can exploit a key characteristic of rhs_sum to do this mapping. Specifically, the rhs_sum will grow monotonically in the order of increasing price values since rhs_sum for a given t.price is the sum of all volume value of records having a smaller or equal price compared to t.price (based on the correlated predicate). Therefore, when a new record arrives, we can first compute the corresponding rhs_sum for that particular record (without considering the new record) and then we know that every rhs_sum greater than or equal to that value must be incremented by the new t.volume. For this, we can define a method shiftkeys(key, offset) that will shift all the keys greater than the given key by the given offset.

Figure 2c shows the final code for this approach. We first compute the rhs_sum for the new record using the getSum method (Line 10). Then, we shift all the keys (i.e., aggregate values) that are greater than or equal to rhs_sum using shiftKeys (Line 13). Since we defined shiftKeys to shift the keys that are *strictly* greater than a given key k, we pass the immediate lesser key (i.e., rhs_sum - volume) to shift all qualifying keys including rhs_sum (Lines 11-13).

Overall, the final code consists of a set of constant time hash map lookups, and linear time getSum and shiftKeys method calls. Hence, the total time complexity is $O(|bids|)$ which is asymptotically faster than DBToaster and re-evaluation that took $O(|bids|^2)$. In the next section, we design a new tree-based index structure called Relative Partial Aggregate Indexes (RPAI) that can perform getSum and shiftKeys operators in logarithmic running time, bringing the overall time complexity down to $O(\log |bids|)$.

3 Relative Partial Aggregate Indexes (RPAI)

In Section 2.2, we saw how the use of PAI Maps that are indexed on aggregate values can improve the incrementalization efficiency of nested aggregate queries. Specifically, for queries with inequality correlated predicates, we identified getSum and shiftKeys as two main operations and showed that PAI Maps can support those operations in linear time. In this section, we design a tree-based data structure called the Relative Partial Aggregate Index (RPAI) that can support both of these operations in logarithmic time. Our data structure needs to be a map (i.e., maintains key-value pairs with unique keys) and should support getSum and shiftKeys in addition to the regular map operations (i.e., get and put).

3.1 Optimizing getSum

Although hash maps support the regular map operations in constant time, getSum and shiftKeys require iterating through all the keys in the map. Alternatively, tree-based data structures are known to be efficient for operations similar to getSum. For instance, Segment Trees [8] and Binary Indexed Trees (Fenwick Trees) [10] can perform range sum queries in logarithmic time which require finding the sum of all elements between two given indices.

We follow a similar intuition and augment a typical TreeMap data structure [30] to maintain the required information in the nodes of the tree. Specifically, we create a Binary Search Tree (BST) indexed by the keys of the map and in each node, store the sum of values of its subtree in addition to storing respective value. Now, we can perform getSum in logarithmic time (assuming the tree is balanced) by recursively traversing the tree leveraging the BST property and calling getSum recursively on the corresponding subtrees.

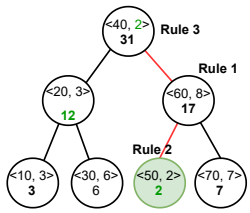


Figure 3: An example run of the getSum(50). The red lines show the traversal path, and the values in green color contributes to the final answer which is 12+2+2. Each node shows <key, value> and the subtree sum.

Figure 3 shows an example run of the algorithm where it finds the sum of all values having keys less than or equal to 50 (i.e., getSum(50)). Having access to subtree sums avoids exhaustive iteration of nodes of subtrees that are known to have keys less than or equal to the given key. For example, in Figure 3, the algorithm avoids traversing the left subtree of the root node by directly extracting the corresponding subtree sum. This enables the algorithm to perform the complete getSum operation in logarithmic time.

3.2 Optimizing shiftKeys

shiftKeys(key, offset) shifts all the keys > key by the given offset (can be negative). Performing this operation on the tree-based data structure is not as trivial as PAI Maps where we iterate through the keys and update the qualifying keys. The complexity comes from the need to ensure that the BST property is intact whenever keys are updated. For cases where offset > 0, we can simply update all qualifying keys since shiftkeys shifts all the keys greater than key without the need for restructuring. That is, the resultant tree (after updating) is guaranteed to satisfy the BST property because (1) for the keys that get updated, they will be shifted by the same value, and (2) for the remaining keys, the set of keys that got incremented were originally greater than them. For cases where offset < 0, special care is needed as the tree structure may need to be changed after the update (discussed in Section 3.2.3). Nonetheless, the overall asymptotic time complexity remains at $O(n)$ since it still needs to update all the nodes with satisfying keys.

3.2.1 Parent-Relative Keys To improve the time complexity beyond $O(n)$, we must design an approach that does not require visiting each qualifying node to perform the update. For that, rather than simply storing the respective key in the node, we can augment the tree structure so that each node stores the key relative to its parent. Specifically, in this setting, the actual key of a node will be the sum of keys along the path from root to the current node. With such a structure in place, changing the key of a node is equivalent to updating the keys of the entire subtree rooted by that node (e.g., incrementing root key by 1 is equivalent to incrementing all keys by 1). Note that this changes the semantics of the raw keys stored in the node, therefore, we need to update the operations (i.e. get, put, and getSum) to take this change into account. Specifically,

Algorithm 1: Shift Keys Operation (for d > 0)

Result: All the keys > k increased by d

```
1  def shiftKeys(node, k, d):
2    if (k < node.key):
3      node.left = shiftKeys(node.left, k - node.key, d)
4      node.key += d
5      node.left.key -= d
6
7      node.minKey -= d
8      node.maxKey -= d
9    else:
10     node.right = shiftKeys(node.right, k-node.key, d)
11   node.minKey = node.left.minKey + node.key
12   node.maxKey = node.right.maxKey + node.key
13   return node
```

this is done by replacing the key in any recursive call with (key - node.key) (similar to lines 3 and 10 in Algorithm 1).

3.2.2 `shiftKeys` *for Positive Offsets* Now that we have a way to update the value of multiple nodes in constant time (ignoring traversal time), we need to design an algorithm that performs `shiftKeys` better than $O(n)$ time. Since negative values can trigger changes in the tree structure which can be nontrivial to handle, we will first look at the case where the given offset value is positive. Algorithm 1 shows the implementation for the `shiftKeys` operator for that case (ignore `minKey` and `maxKey` related operations, we will get back to them later). The algorithm relies on the basic BST property. That is, if a node has a key greater than or equal to the given key, then that node and all the keys on the right subtree need to be incremented with the given offset (lines 2-5). However, unlike a normal tree where we would traverse the entire subtree to update the keys individually, we can simply update the key of the subtree root (line 4). This update also indirectly shifts all the keys of the left subtree which has already shifted the qualifying keys (because of the function call in line 3). Therefore, it is important to make sure that this is corrected by updating the key of the left node (line 5).

Figure 4 shows an example run of the Algorithm 1. Specifically, it runs `getSum(k=9, d=10)` on the input tree where all keys strictly greater than 9 is shifted by 10. This example demonstrates the benefit of storing keys relative to their parent. For instance, notice that all keys on the right subtree of the node with `key=13` are shifted without visiting any of the nodes individually. The overall time complexity of this approach is $O(\log n)$ which is an improvement over the complexity of the PAI Map data structure, $O(n)$.

3.2.3 `shiftKeys` *for Negative Offsets* As discussed before, handling the case where the `offset` is negative is tricky due to the fact that the tree structure may have to change after shifting. For example, if the right child of a node gets shifted by a large negative value such that it is no longer greater than the parent, the resulting tree will not hold the BST property.

We follow a similar intuition to Algorithm 1 to come up with an algorithm for the negative offset case. Specifically, we traverse down the tree and perform the key updates as before. However, after every update, we explicitly check for the BST property and fix the tree if it is violated. For that, in addition to the attributes we have discussed above, we maintain two more attributes in each node. Those are `minKey` and `maxKey` which represent the minimum

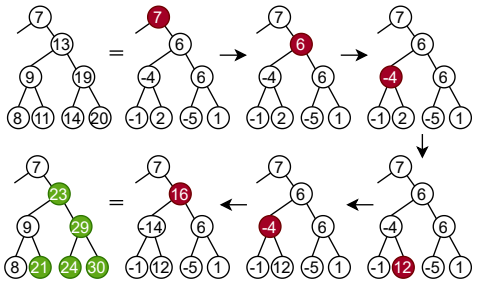


Figure 4: Example run of the Algorithm 1 which runs `shiftKeys(k=9, d=10)` on the given graph. This increments all the keys > 9 by 10. Two leftmost trees are not parent-relative trees and just shown for clarity.

and the maximum keys present in the subtree rooted by the corresponding node. Note that these are based on the parent relative keys (not the actual keys). Although we do not use these attributes when performing `shiftKeys` with positive offset values, we need to make sure that we correctly maintain them during updates to the keys (Lines 7-8 and 12-13 in Algorithm 1).

Algorithm 2 shows the implementation for the `shiftKeys` with a negative offset value. The code looks almost the same as the positive offset case except for the included checks to ensure the BST property is preserved (lines 8 and 12). The first case is when the current node has a key greater than the given k. In that case, the current node and all the nodes in the right subtree will get updated by the same offset value, hence, no violation arising from the right side. However, the left subtree may contain one or more nodes that are less than or equal to the given k, meaning their original keys remaining unchanged. Therefore, there is a possibility that those unchanged nodes having a larger key compared to the subtree root of which the key got decreased, violating the BST property. In our algorithm, this is detected by checking the `maxKey` of the left subtree and calling `fixTree` if the BST property is violated. The same intuition applies to the second case.

Algorithm 2 lines 18-25 shows the pseudo-code for the `fixTree` operator for fixing a subtree having a left subtree that violates the BST property. First, the subtree branch causing the violation of the BST property is removed from the original tree which results in a proper BST. Then, we simply iterate through all the removed nodes and re-insert them into the tree using the `add` operation. The semantics of `add` guarantees that the newly inserted nodes will be put into correct locations. The implementation for the right-side case follows the same idea (omitted due to space limitations).

Figure 5 shows how this algorithm works for an example input. We use actual keys in the figure instead of the parent-relative values to improve clarity. In this example, we shift the largest key of the tree by a large negative value so that the algorithm invokes `fixTree` at each step of the way up. First, `shiftKeys` will recursively traverse the tree to identify the keys that need to be shifted. Then, it shifts the key 5 and then returns back to key 19. Now, the right subtree contains a smaller key (i.e., 5 < 9) and hence, violates the BST property at that node. Therefore, `fixTreeFromRight` is invoked where the right subtree (in this case just 5) is removed and then re-inserted back to the tree. This continues to happen until the root at which the resulting tree becomes correct.

Algorithm 2: Shift Keys Operation (for d < 0)

Result: All the keys > k shifted by d

```
1  def shiftKeys(node, k, d):
2    if (k < node.key):
3      node.left = shiftKeys(node.left, k - node.key, d)
4      node.key += d
5      node.left.key -= d
6      node.minKey -= d
7      node.maxKey -= d
8      if (node.key <= node.left.maxKey + node.key):
9        return fixTreeFromLeft(node)
10   else:
11     shiftKeys(node.right, k - node.key, d)
12     if (node.key >= node.right.minKey + node.key):
13       return fixTreeFromRight(node)
14   node.minKey = node.left.minKey + node.key
15   node.maxKey = node.right.maxKey + node.key
16   return node
17
18 def fixTreeFromLeft(tree):
19   leftSubtree = tree.left
20   tree.left = None
21   tree.minKey = tree.key
22   tree.sum -= leftSubtree.sum
23   for (key, value) in leftSubtree:
24     tree.add(key, value)
25   return tree
```

In fact, this is the worst-case scenario for this algorithm. That is, this example requires traversing to the leaf level of the tree and performs `fixTree` at every level. To analyze the time complexity of the algorithm, we consider a tree with n nodes. Consider a similar scenario as the above example. First, one node will be inserted into a tree with two nodes. Then, three nodes will be inserted into a tree with four nodes and so on. Therefore, we can derive the total time complexity as follows[1] (assuming balanced trees):

$$1 \times \log 2 + (1+2) \times \log 4 + (1+2+2^2) \times \log 8 + \ldots$$
$$= \Sigma_{i=1}^{\log n}(2^i - 1) * i = O(n \log n)$$

Therefore, the overall time complexity of `shiftKeys` comes to $O(n \log n)$ for the negative offset case. This is not ideal since our PAI Map based approach can perform this operation in $O(n)$ time.

3.2.4 Special Case for `shiftKeys` *with Negative Offsets* This worst-case time complexity is applicable for the general `shiftKeys` operation with an arbitrary offset value. However, in our context, we use this data structure to index aggregate values. For instance, in Example 2.2, we index the `rhs_sum` which corresponds to the summation of `volume` values of the `bids` relation. Moreover, we rely on `shiftKeys` when these aggregate values change as a result of tuple insertion (positive offset) or deletion (negative offset). A key realization is that, due to the monotonic nature of the aggregates, deletion of a tuple can only make two aggregate values (i.e., keys in the index) equal in the worst case. Therefore, we can simply extract the corresponding value and then delete the duplicate node from the right subtree. This can be done using the `delete` operation in the normal BST data structures that takes $O(\log n)$ time. Therefore, although the worst-case time complexity of the general `shiftKeys` is $O(n \log n)$, in the context of applying that to optimize nested aggregate queries, it becomes $O(\log n)$.

[1]Summation was solved using `https://www.wolframalpha.com/`

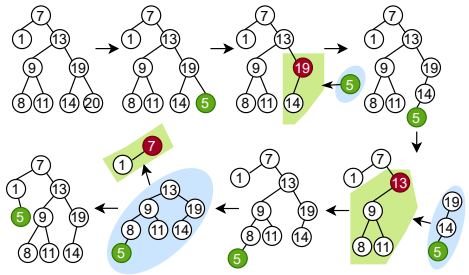


Figure 5: Example run of the Algorithm 2 for `shiftKeys(k=19, d=-15)` which represents the worst-case input. Actual keys are shown instead of parent-relative values to improve clarity. Green node is the node that gets its key shifted. Red nodes are places where `fixTree` is called for BST property violation. In such cases, nodes in the blue background are removed and re-inserted to the subtree in the green background.

3.2.5 Balanced Trees Whenever tree-based data structures are used in practice, it is vital to make the trees balanced to achieve good runtime performance (e.g., TreeMap implementation in Java uses Red-Black Trees [13]). Similarly, we implement a Left-Leaning Red-Black Tree [35] structure and make the necessary changes to the relevant operations (e.g., rotations) to make sure our added attributes (e.g., `minKey`, `maxKey`, `sum`) are maintained properly.

In summary, in this section, we have designed a new tree-based index structured called Relative Partial Aggregate Index (RPAI) trees that supports all the required operations `get`, `put`, `add`, `getSum`, and `shiftKeys` in logarithmic time. We note that we used binary trees in our discussion and implementation, but the same principles would apply to B-trees [3] as well.

4 Incrementalizing Algorithm and Implementation

In the previous section, we have introduced PAI Maps and RPAI Trees and demonstrated how building such index structures on aggregates can improve the performance of correlated nested aggregate queries. In this section, we generalize our approach and present algorithms for incrementalizing different types of aggregate queries with correlated nested subqueries in their join predicates.

4.1 Supported Queries

We specifically target aggregate queries that contain correlated or uncorrelated aggregate subqueries in their join predicates. We construct a formal grammar (shown below) to represent such queries in a concise manner instead of relying on the general relational algebra which might make the representation more verbose.

$$\begin{aligned}
AggrQ &\rightarrow Aggr_{[cols]}(AggrFunc, Relations, Predicates)\\
AggrFunc &\rightarrow AggrFunc\ \boldsymbol{op}\ AggrFunc\\
AggrFunc &\rightarrow (SUM|COUNT|AVERAGE|MIN|MAX)f(cols)\\
Relations &\rightarrow Relation \mid Relation, Relation \quad Relation \rightarrow Q \mid R\\
Predicates &\rightarrow Predicate \mid Predicate\ (AND|OR)\ Predicate\\
Predicate &\rightarrow Value\ \boldsymbol{\theta}\ Value \quad \boldsymbol{\theta} \rightarrow\ > \mid >= \mid < \mid <= \mid =\\
Value &\rightarrow Value\ \boldsymbol{op}\ Value \quad \boldsymbol{op} \rightarrow\ +|-|\times|\div\\
Value &\rightarrow Const \mid Col \mid Aggr_{[]}(AggrFunc, Relations, Predicates)
\end{aligned}$$

Any given query mainly contains four parts. First, there is a top-level aggregate function that corresponds to the final aggregate value that needs to be computed. This can be any arbitrary function that uses aggregates of any computations on columns (e.g., `SUM(r.A) + COUNT(r.B)`). Second, there is a set of relations (base tables or other subqueries) that are joined in the query. Third, there is a set of predicates (that are connected by `AND` or `OR`). Each predicate can contain nested aggregate queries, constant values, or columns from one of the join relations. Finally, the query specifies a set of group-by columns for which the final aggregate values need to be grouped. Shown below is the query in Example 2.2.

$$q1 = Agg_{[]}(SUM(b.price \times b.volume), (bids\ b), q2 < q3)$$
$$q2 = Agg_{[]}(SUM(b1.volume), (bids\ b1), \emptyset)$$
$$q3 = Agg_{[]}(SUM(b2.volume), (bids\ b2), b2.price \leq b.price)$$

We also define a utility function $free$ that finds all the columns that are referred within q and that are not from relations used inside the query ($free_r$ represents the subset corresponding to relation r). That is, in the case of correlated subqueries, this will be the set of correlated columns. Similarly, we define *bound* to retrieve the rest of the columns used in predicates. For example, in the query above $free_{bids}(q1) = \emptyset$, $free_{bids}(q3) = \{price\}$, $bound_{bids}(q1) = \emptyset$, and $bound_{bids}(q3) = \{price\}$. Then, we define another utility $extractPredVals$ function that extracts the predicate values, given a query. For example $extractPredVals(q1) = \{q2, q3\}$.

We only focus on aggregate queries that may contain nested aggregate queries in their join predicates since these are the type of queries other systems fail to handle efficiently (as discussed in Section 1). For cases where such queries occur as subqueries of other larger queries, our approach can be used incrementalize the respective portion whereas the rest of the query can be handled by already existing approaches (e.g., DBToaster).

4.2 General Incrementalization Algorithm

We first look at a general incrementalization algorithm that works for arbitrary queries and has better asymptotic runtime properties compared to existing approaches. In the subsequent sections, we will look at several specific commonly occurring query patterns where we further optimize using PAI Maps and RPAI Trees. We will use the query in Example 2.2 to illustrate each step.

4.2.1 Execution Model We follow a similar approach to DBToaster for the query execution model. Specifically, given a query and a set of base relations (i.e., data stream sources), we first identify a set of maps that needs to be maintained to compute the final result incrementally. Then, we construct update triggers for each relation that would maintain those maps upon new tuple arrivals (or deletions). Whenever a new tuple arrives, the corresponding trigger will be called and the final result is computed after updating the indexes (similar to Figure 1c and 2c).

4.2.2 Initializing the required maps Although we follow a similar execution model to that of DBToaster, we do not rely on delta rules (that does not work for nested aggregates) to construct the required maps. Instead, we come up with a new algorithm to find the required indexes for queries with nested aggregates.

Our algorithm works on a simple intuition. For an aggregate query with correlated subquery predicates, whenever a new tuple arrives, two things happen. First, if the base relation corresponding to the new tuple is used in the outer query, then, there will be a new tuple for the outer relation. Hence, the predicates must be evaluated for this new tuple and if it satisfies the predicates, it should be considered in the final result. For instance, for VWAP (Example 2.2) an insertion to *bids* will result in a new tuple for the outer *bids* b relation for which the predicates must be evaluated. Second, the addition of this new tuple affects the nested aggregate predicate values of other outer tuples, hence, affecting their predicate evaluations. For example, in VWAP, insertion to *bids* will result in change of aggregate value of the left-side (because of new tuple in `b1`) and the right-side (because of new tuple in `b2`). Therefore, now for each tuple in the outer relation `b`, both sides of the predicate may have been changed, hence, need to be reevaluated to construct the result.

For any predicate value that is a constant or an uncorrelated nested aggregate, the value will be fixed for all the tuples, hence, can be computed independently. For example, the left side of the VWAP query will be a fixed value for all the outer tuples. Moreover, in the case where the value simply refers to a computation of columns from the outer tables, tuples from the outer relation would not have an impact on the other tuples. However, in the other case, where the nested aggregate query is correlated to the outer query, the addition of new tuples could change predicate evaluations of other outer tuples as well. We need to maintain index structures to efficiently keep track of these changes and compute the result without recomputing from scratch. Specifically, for each of the correlated nested aggregate predicates, we create two separate maps that map to the nested aggregate value from free and bound columns separately. Moreover, we construct a set of indexes that maps from the union of bound and free variables (off all predicate subqueries) to the final result aggregate. In the subsequent subsections, we will see how these maps are used to compute the final result. For any nested aggregate predicate value that contains multi-level nesting, we apply this approach recursively and initialize the relevant maps.

Consider the VWAP query. The left side in the predicate is not a correlated query, therefore the aggregate value can be independently maintained (e.g., in variable `lhs_sum`). For the right side aggregate subquery, both free and bound variables are the same $\{price\}$. Therefore, we need to create two maps mapping price to the aggregate sum (e.g., `freeMapRhs: price` $\rightarrow$ `rhs_sum` and `boundMapRhs: price` $\rightarrow$ `rhs_sum`). Finally, we need a map that maps $free \cup bound$ to the final aggregate value (e.g., `resMap: price` $\rightarrow$ `resSum`). These are shown in Figure 6, lines 1-4. We will walk through the rest of the code and observe how these maps are maintained and used in the next subsection.

4.2.3 Creating Update Triggers Once the required maps are constructed, we need to generate triggers for each base relation that maintain these maps under updates. Algorithm 3 shows how such a trigger is created for relation R_i. First, we iterate through free and bound column values of all relations except R_i. This is done because the predicate values (hence, the respective indexes) of all those relations only depend on the respective free and bound column values. Then, there are mainly three maps that need to be updated.

First, the addition of a new tuple changes the bound maps of any predicate that has columns from R_i as bound variables. Lines 8-11 show how this update is performed. Specifically, we iterate through all the predicate values that use columns from R_i and check whether it is a correlated subquery. Note that this and other similar

Algorithm 3: Creating the update triggers

```
1  def trigger_Ri(t):
2   for vR1 in freeAndBoundIters[R1]:
3    for vR2 in freeAndBoundIters[R2]:
4     # skip Ri
5     ....
6     for vRn in freeAndBoundIters[Rn]:
7      values = {vR1, vR2, .., vRn}
8      for Vi in extractPredValues(Q, Ri):
9       if (isCorrelatedAggr(Vi)):
10       # update the bound maps
11       predBoundMaps[Vi][Ri].add(t,
12                                 Vi.aggrSum(values, t))
13      # update affected aggr values
14      for key in predFreeMaps[Vi][Ri].get(values):
15       if Vi.evaluatePredicate(values, t, key):
16        predFreeMaps[Vi][Ri].add(values + key,
17                                 Vi.aggrSum(values, t, key))
18      # find the aggr value for the newTuple
19      if not predFreeMaps[Vi][Ri].exists(values + t):
20       aggr = 0.0
21       for key in predBoundMaps[Vi][Ri].get(values):
22        if Vi.evaluatePredicate(values, key, t):
23         aggr += Vi.aggrSum(values, key, t)
24       predFreeMaps[Vi][Ri].update(values + t, aggr)
25     # update the result maps
26     for reqRes in resMaps[Ri]:
27       resMaps[Ri][reqRes].add(values + t,
28                               compute(reqRes, t))
```

checks have to be done only once (i.e., during trigger generation) as at runtime, the constructed triggers are specialized for a given query. Then, we retrieve the corresponding map and update the relevant entry. Here, we directly pass `t` as a key to `add` and omit the code for extracting the correct key (i.e., values of bound columns in this case). For VWAP (Figure 6), `boundMapRhs` is updated in line 9.

Second, we need to update the aggregate values (in free maps) that are affected by the addition of the new tuple to the inner relations (shown in lines 14-17). We iterate through the keys in the corresponding free map and evaluate the predicate for the new tuple. If it satisfies the predicate, the aggregate value is affected, hence, the map is updated. Here, we pass `key` and `t` for the values of free and bound columns respectively as we are updating the change caused by the addition of the new tuple to the inner relation (i.e., bound). For VWAP, this is done in lines 11-13 in Figure 6.

Next, we need to compute the aggregate value for the new tuple in the outer relation. This can be computed by iterating through the keys in the corresponding bound maps and evaluating the predicate and accumulating the aggregate value (lines 19-24). Here, we should pass `key` and `t` for the values of bound and free columns respectively (i.e., opposite to the above case) because now we are computing the aggregate value for the new tuple in outer relation (i.e., free in the context of subquery). Figure 6 lines 15-20 shows this for VWAP.

Finally, we need to update all the result maps corresponding to the relation R_i. This can be done by simply iterating through all the result maps of R_i and updating the maps with the corresponding value (Lines 28-30). Here, `values + t` refers to appending `t` to `values`. For the VWAP query, this is done in line 22 in Figure 6.

4.2.4 Computing the final result After updating all the maps, the next step is to compute the final result using the updated maps. For this, we iterate through all the free and bound column values for

```
1  lhs_sum = 0          # maintains lhs_sum
2  freeMapRhs = Map()   # price -> rhs_sum
3  boundMapRhs = Map()  # price -> Sum(volume)
4  resMap = Map()       # price -> Sum(price * vol)
5  def on_new_bids(t):
6   # update lhs_sum
7   lhs_sum = 0.75 * t.X * t.volume
8   # update bound maps
9   boundMapsRhs.add(t, t.X*t.volume)
10  # update affected aggr values
11  for price in freeMapRhs:
12   if t.price <= price:
13    freeMapRhs.add(price, t.X*t.volume)
14  # find the aggr value for the new tuple
15  if not t.price in freeMapRhs:
16   aggr = 0.0
17   for price in boundMapRhs:
18    if price <= t.price:
19     aggr += boundMapRhs.get(price)
20   freeMapRhs.update(t.price, aggr)
21  # update resMap
22  resMap.add(t.price, t.X*t.price*t.volume)
23  # compute the result
24  res = 0.0
25  for price in resMap:
26   if lhs_sum < freeMapRhs.get(price):
27    res += resMaps.get(price)
28  return res
```

Figure 6: Code generated for incrementally computing the VWAP query (Example 2.2) using the General Incrementalization Algorithm (Section 4.2)

all the relations and evaluate the query predicate. When evaluating the nested aggregate values, we can directly use the free maps that we have created before to get the corresponding aggregate value without doing any more computations. Then for keys that satisfy all the query predicates, the corresponding results maps are queried to get the aggregate sum. Figure 6, lines 26-30 shows how the final result is computed for the VWAP query. The overall time complexity of this approach is $O(n^k)$ where n is the cardinality of relations and k is the number of relations in the query. In comparison, DBToaster can take up to $O(n^k \times n^k)$ for processing queries of this structure.

4.2.5 Limitations The algorithms discussed above assumes the aggregates functions are *streamable* [16]. That is, the aggregate values can be updated by having information about the current aggregate and the new value (e.g., `SUM`, `COUNT`, `AVERAGE`, etc.). For other types of aggregates (e.g., `MIN/MAX`), our approach of storing the aggregate would only work for insertion-only updates. For deletions, we cannot recover the updated aggregate value for such aggregate types since it is impossible to recover the new value with just the previous aggregate value. One way to handle this is to keep a binary search tree of the data instead of storing just the aggregate value. Now, for tuple deletions, we can simply remove the corresponding value from the tree and retrieve the next maximum or minimum value in logarithmic time (assuming balanced trees). However, it is not clear how this can be generalizable beyond `MIN/MAX` to any arbitrary non-streamable monoid.

4.3 Partial Aggregate Index Optimization

The general algorithm above can improve incrementalization efficiency for a wide class of aggregate queries that contain correlated nested aggregates as predicates. In this section, we look at how we

Algorithm 4: Creating the update triggers

```
1  def trigger_Ri(t):
2   tAggrVal = QRi.aggrVal(t)
3   if predicateType(QRi) == "=":
4    newAggr = boundMaps[Ri].get(t) + tAggrVal
5    affectedAggr = freeMaps[Ri].get(t)
6    # shift point aggregates
7    for reqSum in requiredSums(Q, Ri):
8     valToMove = resMaps[Ri].get(t)
9     aggrMaps[Ri].add(affectedAggr, -valToMove)
10    aggrMaps[Ri].add(affectedAggr+tAggrVal, valToMove)
11    aggrMaps[Ri].add(newAggrSum, reqSum(t))
12    resMaps[Ri].add(t, reqSum(t))
13
14  elif predicateType(QRi) == "<=":
15   newAggr = boundMaps[Ri].getSum(t)
16   affectedAggr = freeMaps[Ri].getSum(QRi.boundComp(t))
17   # shift all aggr >= affectedAggr
18   for reqSum in requiredSums(Q, Ri):
19    aggrMaps[reqSum].shiftKeysInclusive(
20                              affectedAggr, reqSum(t))
21    aggrMaps[reqSum].update(newAggr, reqSum(t))
22  elif predicateType(QRi) == "<":
23   ...
24  ...
25  # update the rest of the maps
26  freeMaps[Ri].add(t, tAggrVal)
27  boundMaps[Ri].add(t, tAggrVal)
```

can further optimize certain classes of nested aggregate queries that appear commonly in practice (e.g., computing useful metrics in algorithmic trading [22]) using PAI Maps and RPAI Trees introduced in Section 3. As discussed in Section 2, a key idea to improve the incrementalization efficiency of nested aggregate queries is to build indexes that are indexed by aggregate values. Then, whenever a new tuple arrives, shift a single or a range of aggregate values (i.e., keys in these trees) that are affected and then use these updated indexes to compute the final result.

We do note that this approach only works for queries having a certain structure compared to the general algorithm which works for any general query. The main requirement for this approach to work efficiently is that the addition of a new tuple should only affect a single aggregate value or a single range of aggregate values per aggregate index. Whenever a range of aggregates needs to be shifted, we can use the `shiftKeys` method introduced in Section 3. For a query that is operated on a single base table (or subquery) R_1, it should have the following structure to use aggregate indexes to improve the incrementalization.

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

Here, the $AggrQ$ should be a streamable aggregate function, v_1 is either a *const* or an $AggrQ$ with $free(v_1) = \emptyset$ (hence, can be independently maintained), and q_1 is an $AggrQ$. Note that v_1 and q_1 can be on opposite sides as well. A key characteristic of this query is that, for all the tuples from the outer relation, v_1 will have the same value (because not correlated). Therefore, if there is an aggregate index that maps the aggregate values of q_1 to the aggregate values required for the final query results, we can directly compute the query result. For example, if θ was '=', then we get the value of v_1 and query the aggregate index with the value of v_1 to get the result. For other cases, we can use `getSum` method to find the final aggregate values that are within the range implied by the predicate.

To maintain this aggregate index, the addition of a new tuple should change either a single aggregate value or a range of aggregates. This happens if either q_1 contains multiple conjunctive equality predicates (results in a single point update) or q_1 contains a single inequality predicate (results in an update of range of aggregates). For example, in the VWAP query (Example 2.2), predicate in the nested query is the inequality `b2.price <= b.price`, meaning that whenever a new tuple arrives for `b2`, all the aggregates values that have a `price` greater than the `price` of new tuple must be updated (by the same amount).

Shown below is the structure of queries with multiple relations that supports the aggregate index optimization.

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

Here, $AggrFunc$ and $v_1, \ldots v_n$ has the same properties as before, and for q_{R_i}, it should be correlated only on columns from R_i (i.e., $free(q_{R_i}) \subseteq R_i.columns$). Similar to the general algorithm, we create free and bound maps for each of the correlated nested subqueries. Moreover, we create aggregate indexes that maps aggregate values (i.e., q_{R_i} values) to the required result sums.

Algorithm 4 shows how the trigger is generated for relation R_i. The code generated for each relation depends on the type of subquery predicates. The pseudocode only shows the case for equality and $\leq$ case, and rest of the cases follow a similar strategy. The crux of the algorithm is identifying the single aggregate value or range of aggregates that need to be shifted and performing the update.

4.3.1 Overhead of Identification For our approach to be applied to a general incremental query processing setting, it is imperative to be able to recognize the queries that our optimizations can be applied. Given a query (or part of another query), we can first check whether the query computes an aggregate. Then, we can make use of the helper functions $free$ and *bound* to determine if our query fits the structure for applying the aggregate index optimization. Therefore, identification can be done as part of the query optimizer (e.g., pattern matching on the query tree) and would have a similar overhead. Once the eligible queries have been selected, there is a cost associated with creating the required maps and generating the triggers. Similar overheads exist for other systems that follow a similar execution model. For example, identifying the different predicates and their value types (e.g., Line 9-10 in Algorithm 3) is done in trigger generation time and would not have an impact on the runtime performance. Overall these initializations take a time linear to the size of the query (i.e., no exponential blowup).

4.3.2 Limitations This approach only works for the queries that are in the structure mentioned above, and cannot apply generally for other types of aggregate queries. Moreover, since this optimization relies on indexing maps with aggregate values and shifting aggregate ranges, this only works with aggregates like `SUM`, `COUNT`, and `AVERAGE` and cannot be used for `MIN/MAX`.

4.4 Prototype Implementation

We implement a prototype of our approach in a high-level programming language, Scala. Scala's features such as first-class functions, case classes, and pattern matching make the implementation and management of the different types of trigger functions relatively convenient. Moreover, using existing techniques like Multi-Stage Programming (MSP) [32, 33, 36], we can eradicate the overhead

Query	Optimizations		DBToaster	RPAI (Ours)
	GA	Aggr		
MST, VWAP, NQ1	✓	✓	$O(n^2)$	$O(\log n)$
PSP	✓	✓	$O(n)$	$O(\log n)$
SQ1, SQ2	✓	✗	$O(n^2)$	$O(n)$
NQ2	✓	✗	$O(n^3)$	$O(n \log n)$
TPC-H Q17	✓	✓	$O(n)$	$O(\log n)$
TPC-H Q18	✓	✗	$O(1)$	$O(1)$

Table 1: Queries used for evaluation with their corresponding optimizations and asymptotic time complexity. GA - general algorithm, Aggr - aggregate index optimzation

of using such high-level features by generating specialized native code. This approach generally achieves orders of magnitude performance improvements [9, 31, 37]. However, this performance optimization is beyond the scope of this paper, and hence, we only use the techniques presented in this paper with the corresponding Scala implementation for experiments in Section 5.

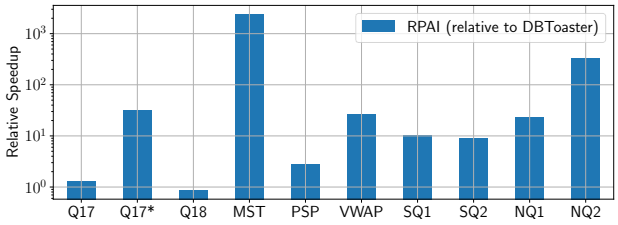
Our implementation is done in the context of an in-memory incremental processing setting where we assume there is sufficient memory to hold the indexes and any other data required. We rely on the JVM for memory management. If this memory usage becomes a bottleneck, we can use techniques like MSP to compile our program into a specialized low-level code where we can rely on the operating system (i.e., paging) or custom memory management mechanisms. Moreover, the ideas presented in this paper are not limited to the context of processing data streams but are generally applicable to any other incremental processing use cases (e.g., IVM in DBMS, approximate query processing [46], etc.).

5 Experimental Evaluation

In the previous sections, we have demonstrated how to use RPAI Trees and PAI Maps to achieve better asymptotic time complexities for queries with correlated nested aggregates. In this section, we run several benchmarks and measure the actual execution time for workloads of practical importance to ensure that the expected performance characteristics are indeed realized in practice.

5.1 Benchmark Environment

5.1.1 Data and Query Workload There is no single established benchmark for evaluating the performance of correlated nested aggregate queries specifically. Therefore, we use two existing datasets with the addition of synthetic queries for our evaluation. The first workload consists of running algorithmic trading related queries on a stream of order book data (used in prior related work [24, 25]). There are two types of order book entries; *bids* and *asks* each having *timestamp*, *broker_id*, *price* and *volume* as their attributes. We evaluate the performance for MST, PSP, and VWAP queries from the original benchmark and added SQ1, SQ2, NQ1, and NQ2 synthetic queries to evaluate different types of correlated nested aggregate queries. The second workload is using selected queries from the TPC-H benchmark adapted to an incremental processing setting (similar to [24, 29]). We specifically focus on Q17 and Q18, which contain nested aggregates.



Time (ms)	Q17	Q17*	Q18	MST	PSP	VWAP	SQ1	SQ2	NQ1	NQ2
DBToaster	42864	1097090	42410	168520	88	888	3967	1265	2957	202544
RPAI(Ours)	33066	35041	46417	70	32	34	379	140	129	610

Figure 7: Relative execution time for queries compared to DBToaster. All queries except TPC-H Q18 have better performance.

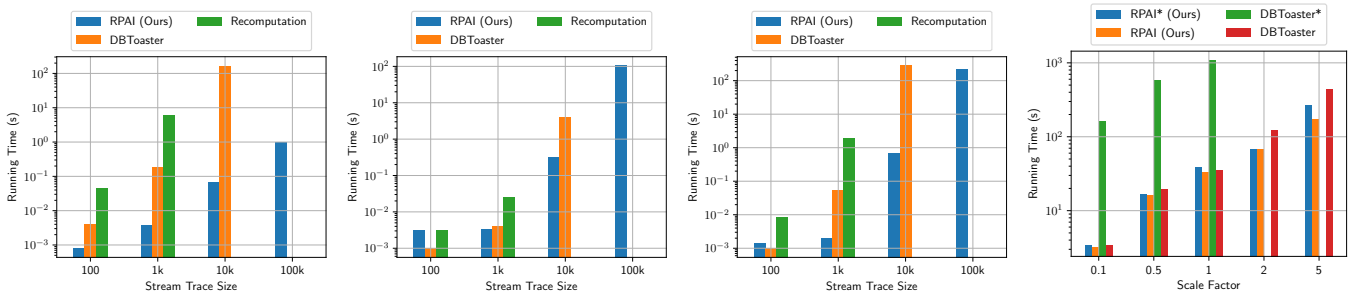
5.1.2 System Environment We run all our experiments on a NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total) running Ubuntu 18.04.4 LTS. We use DBToaster 2.3 and use the generated Scala code in our experiments. Our implementation uses Scala 2.10.4. All the experiments are single-threaded and run on a single-core. We use default JVM configurations for most cases except for TPC-H queries where we increase the max heap size to per-socket memory.

5.2 Evaluating Incrementalizing Performance

5.2.1 Query Properties Table 1 summarizes the specific optimizations we perform on each query and the time complexity difference between our approach and DBToaster. MST and PSP are operating on both *asks* and *bids* and perform a cross join with inequality predicates. MST has four nested aggregates of which two are correlated. As DBToaster does not incrementalize correlated nested aggregates, it needs to iterate through records from both relations to compute those correlated subqueries. We incrementalize those queries using RPAI Maps. Therefore, whenever a new record arrives, we can compute the corresponding aggregate value and shift all the affected aggregate values in logarithmic time (as demonstrated in Section 4). PSP is similar to MST but contains join predicates on a column (`volume`) instead of a correlated nested aggregate. We follow a similar approach to incrementalize PSP in logarithmic time.

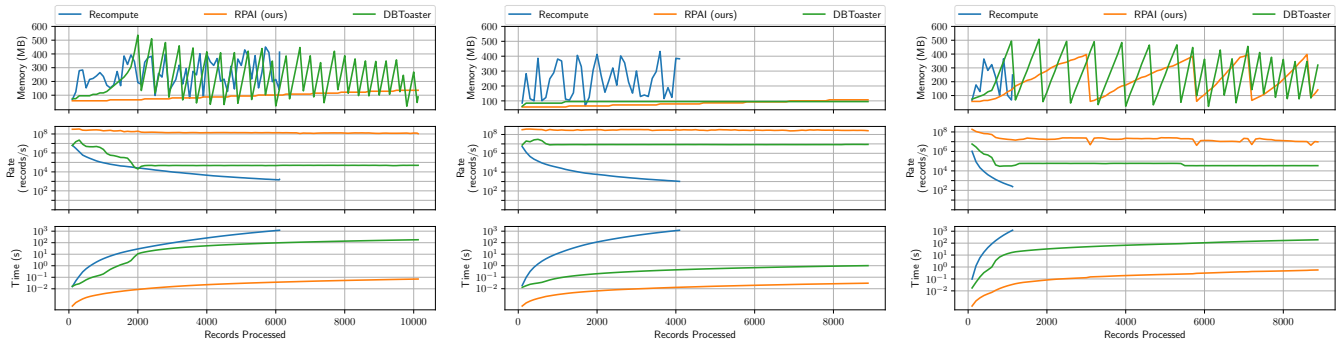
We saw VWAP in Example 2.2. It contains two nested aggregates; one correlated with the outer query. SQ1 and SQ2 are modified versions of the VWAP query. SQ1 makes the uncorrelated subquery into a correlated one by adding a predicate inside the nested subquery. With this modification, both sides of the predicate become variable for a given outer record. Hence, the final result computation can no longer be done in logarithmic time using `getSum` operator and has to rely on the general algorithm (hence, $O(n)$)). SQ2 changes the right-side inner predicate of VWAP to contain asymmetric computations in the two sides of the inequality.

NQ1 and NQ2 are also modified versions of VWAP that replace the nested aggregate with another multi-nested aggregate query. Specifically, NQ1 replaces the nested aggregate in VWAP with another correlated nested aggregate query like VWAP (i.e., containing 2-level nesting). NQ2 is similar but the replaced query also has a correlation to the outer query (i.e., the lowest level is correlated to the outermost query). NQ1 is handled by computing the delta of the new subquery independent of the outer query. Once we compute the delta, the rest of the computation is the same as VWAP



(a) MST (b) SQ1 (c) NQ2 (d) Q17 (*for skewed dataset)

Figure 8: Analyzing the scalability of our algorithm for different queries. (d) contains results for both the uniform (default) and the non-uniform (augmented) versions of the TPC-H datasets.



(a) MST (b) VWAP (c) NQ2

Figure 9: The memory footprint (top), rate of processing records (middle), and time (bottom) as incoming records get processed for three different queries. Rate and Time are in log-scale. Sudden drops of memory usage (with a corresponding decline in rate) are a result of garbage collection. Our implementation has better overall performance compared to the other two cases.

incrementalization. For NQ2, we have to rely on the general algorithm for the outermost query. DBToaster uses three nested loops whereas we incrementalize this query in $O(n \log n)$ time.

TPC-H Q17 and Q18 contain a single correlated and uncorrelated nested-aggregate respectively. We can improve the efficiency of Q17 over DBToaster by using our approach. However, since Q18 is uncorrelated, both our implementation and DBToaster fully-incrementalize the query, leaving the time complexity at the same level. We added Q18 to demonstrate that our approach still achieves competitive performance for other types of aggregate queries.

5.2.2 Performance Analysis Figure 7 shows the relative execution time for all queries. In this experiment, we run all the queries on a trace of stream of the same size (10k records for finance queries, Scale Factor 1 for Q17 and Q18). We observe that most queries show a significant speedup (up to more than 1100 ×) which is in line with the expected performance gains due to lower asymptotic time complexity. Our implementation has a similar performance to DBToaster for Q17 and Q18. This can be partly attributed to the efficiency of specialized internal data structures (e.g., specialized hash-maps) generated by DBToaster. Although it explains the performance of Q18, it still cannot explain why DBToaster scales at a similar rate to our implementation for different sizes of datasets for Q17, despite having a higher time complexity (Figure 8d). Consider Q17:

```
SELECT SUM(l.extendedprice) / 7.0 AS avg_yearly
FROM   lineitem l, part p
WHERE  p.partkey = l.partkey AND  p.brand = 'Brand#23'
  AND  p.container = 'WRAP BOX'
  AND  l.quantity < (
         SELECT 0.2 * AVG(l2.quantity) FROM lineitem l2
         WHERE l2.partkey = p.partkey)
```

Our implementation incrementalizes this by maintaining an RPAI Tree (`quantity` → `avg_yearly`) and the aggregate value of the nested aggregate query for each distinct `partkey`. Whenever, a new `lineitem` arrives, we update the corresponding index and the aggregate value, and compute the change to `avg_yearly` using `getSum(rhs_sum)`.

In the case of DBToaster, whenever a new `lineitem` arrives, first it updates the nested-aggregate sum (similar to us) and it loops over all the `lineitems` that have the same `partkey` and evaluate the predicate and updates the `avg_yeary` change. However, rather than iterating over *all* the line items with the same part key, they are only iterating over line items with unique `quantity` values. This is done by a multi-level index that maintains partial sums for each unique `quantity` per unique `partkey` for `lineitem` records (i.e., `partkey` → `quantity` → `avg_yearly`). This works well for

datasets like TPC-H where the data is uniformly distributed. Therefore, although the worst-case time complexity is O(n), due to the uniformity of the data, the updates can be performed in a very small fraction of the total records. We note that the amount of computation is still lower in our implementation compared to this optimized strategy as our TreeMap also maintains unique `quantity` values. To validate our hypothesis, we analyzed the behavior of DBToaster against a skewed dataset. We augmented the TPC-H data generation framework to generate skewed data. As expected, the performance gap grows from 1.3× to more than 30× (Q17* in Figure 7). Therefore, although this is an extremely useful optimization for performing simple nested aggregates on uniform datasets, it does not work well in other scenarios. Moreover, it only works for nested subqueries correlated on equalities.

Figure 8 analyses the scalability of our approach over the stream trace size. The workload for the first three queries (i.e, MST, SQ1, and NQ2) is selected from different sizes of stream traces from the original finance dataset [24]. For the last figure, we use the TPC-H dataset of scale factors 0.1, 0.5, 1, 2, 5 (100MB, 500MB, 1GB, 2GB, and 5GB respectively), both the uniform and the skewed version. In some cases, for smaller datasets (up to 1k in finance, SF=1 in TPC-H), our approach performs competitively or worse than both recomputation and DBToaster. This can be attributed to the fact that our approach initializes more index structures and for smaller workloads, maintenance of these indexes can outweigh the performance benefits. However, as the workload size increases, our approach performs significantly better and scales well (as expected due to better asymptotic time complexity).

Figure 9 shows how the runtime performance characteristics change as the stream of data gets processed. Specifically, we analyze how the memory usage, rate of processing records, and time varies over the number of records processed. The rate of processing records generally drops over time as more data is manipulated in indexes and iteration spaces become larger. We can see our implementation consistently has a better rate compared to DBToaster due to being able to perform the updates efficiently. We can also observe that for some cases the memory usage contains spikes at regular intervals. These correspond to garbage collection (GC) triggered by the Java Virtual Machine (JVM). This is also evident from the fact that there is a decline in the rate of processing records at the same point in time. The memory usage just after a GC run is closer to the starting memory in many cases, indicating that the working set size does not grow notably over time. For all the queries, we can see that these GC cycles happen more frequently in DBToaster compared to our implementation, implying better allocation rates. We note that these behaviors may only be applicable for languages like Scala where there is a managed runtime and may see different behavior for runtimes with explicit memory management.

6 Related Work

Traditional Incremental View Maintenance (IVM) [5, 6, 12, 14, 15] approaches deal with maintaining materialized views on updates to underlying base tables. Generally, these approaches do not focus on performing real-time analytics on fast data streams, and hence, are not optimized for low latency. DBToaster [22, 24, 29] applies the idea of IVM in a recursive manner [23] to maintain a view using a hierarchy of materialized views. Some prior works present algorithms and data structures for incrementalization of specific types of queries. DYN [16] and IDYN [17, 18] are two approaches that focus on acyclic Conjunctive Queries (CQ) with equality and inequality join predicates respectively and AJU [42] focuses on acyclic foreign-key joins. None of these approaches efficiently incrementalize queries with correlated nested aggregate sub-queries when correlation removal [11] is not possible.

Incremental query processing algorithms are used in many other use cases including approximate query processing [46], progressive data warehouses [43, 44] and intermittent query processing [39]. Some works investigate how to balance the resource consumption for view maintenance and latency [7, 39–41, 47] and improve overall performance in multi-query settings by sharing computations across concurrent queries [20, 21, 26, 41]. However, many of the incrementalization algorithms presented in these works either do not efficiently incrementalize complex queries such as correlated nested aggregates or are restrictive due to their application-specific requirements (e.g., produces approximate results, does not support tuple deletions, etc.). Therefore, such algorithms are not directly applicable to cases where perfectly accurate results are needed to be produced as a stream. Since the algorithms presented in this paper are relatively general, we believe our approach can be applied in these settings to improve their performance.

Stream processing systems [1, 2, 4, 45] process data streams and incrementally maintain the query outputs. These systems are mainly designed for queries with window semantics. Naiad (Timely Dataflow [TD]) [28], Differential Dataflow (DD) (build on top of TD) [27] are two frameworks that allow scalable incremental computation. Materialize [19] is an IVM engine built on top of both TD and DD. We believe that the algorithms and the data structures presented in this paper can be implemented in those systems for efficient incrementalization of nested aggregates.

TreeMap is a data structure found in the standard libraries of many programming languages [30, 34], and of course, tree indexes are standard in any RDBMS [3]. Fenwick Trees [10] and Segment Trees [8] are two tree-based data structures that support operations similar to `getSum` in logarithmic time. However, none of them have support for efficiently shifting key ranges. To our knowledge, the data structure we proposed for RPAI Trees is the first to support both `getSum` and key shifts (`shiftKey`) in logarithmic time.

7 Conclusions

We have introduced a novel algorithm and indexing structures called PAI Maps and RPAI Trees for efficiently incrementalizing complex nested-aggregate queries with correlations. We have presented a new data structure to realize these index structures efficiently and demonstrated the performance gains in both asymptotic time complexity and actual execution time. We believe the ideas presented in this paper can be applied to other incremental execution settings to improve the performance of these types of queries.

8 Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Facebook, Google, Microsoft, and VMware.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. STREAM: The Stanford Data Stream Management System. In *Data Stream Management*. Springer, 317–336.

[3] Rudolf Bayer and Edward M. McCreight. 1970. Organization and Maintenance of Large Ordered Indexes. In *SIGFIDET Workshop*. ACM, 107–141.

[4] 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.

[5] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *ICDE*. IEEE Computer Society, 190–200.

[6] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Found. Trends Databases* 4, 4 (2012), 295–405.

[7] 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

[8] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. 231–236 pages.

[9] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. USENIX Association, 799–815.

[10] Peter M. Fenwick. 1994. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exp.* 24, 3 (1994), 327–336.

[11] César A. Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD Conference*. ACM, 571–581.

[12] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *SIGMOD Conference*. ACM Press, 328–339.

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

[14] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[15] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *SIGMOD Conference*. ACM Press, 157–166.

[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.

[17] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2018. Conjunctive Queries with Inequalities Under Updates. *Proc. VLDB Endow.* 11, 7 (2018), 733–745.

[18] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient Query Processing for Dynamically Changing Datasets. *SIGMOD Rec.* 48, 1 (2019), 33–40.

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

[20] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: Ad-hoc Stream Joins at Scale. *Proc. VLDB Endow.* 13, 4 (2019), 435–448.

[21] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-hoc Shared Stream Processing. In *SIGMOD Conference*. ACM, 607–622.

[22] Oliver Kennedy, Yanif Ahmad, and Christoph Koch. 2011. DBToaster: Agile Views for a Dynamic Data Management System. In *CIDR*. www.cidrdb.org, 284–295.

[23] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *PODS*. ACM, 87–98.

[24] 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.

[25] 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.

[26] 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.

[27] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*. www.cidrdb.org.

[28] 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.

[29] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD Conference*. ACM, 511–526.

[30] Oracle. 2021. *TreeMap (Java Platform SE 8).* Retrieved June 22, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

[31] Tiark Rompf and Nada Amin. 2019. A SQL to C compiler in 500 lines of code. *J. Funct. Program.* 29 (2019), e9.

[32] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136.

[33] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*. ACM, 497–510.

[34] Scala. 2021. *TreeMap (Scala Standard Library).* Retrieved June 22, 2021 from https://www.scala-lang.org/api/current/scala/collection/mutable/TreeMap.html

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

[36] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

[37] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

[38] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. 2000. Progressive evaluation of nested aggregate queries. *VLDB J.* 9, 3 (2000), 261–278.

[39] 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

[40] 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

[41] 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

[42] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *SIGMOD Conference*. ACM, 1225–1239.

[43] 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

[44] 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

[45] 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.

[46] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *SIGMOD Conference*. ACM, 1347–1361.

[47] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. 2007. Lazy Maintenance of Materialized Views. In *VLDB*. ACM, 231–242. https://doi.org/doi/10.5555/1325851.1325881