Integrated Querying of SQL database data and S3 data in Amazon Redshift

Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel
Ippokratis Pandis, Yannis Papakonstantinou, Michalis Petropoulos
{mengchu, magrund, awgupta, fbnagel, ippo, yannip, mpetropo}
@amazon.com

Abstract

Amazon Redshift features integrated, in-place access to data residing in a relational database and to data residing in the Amazon S3 object storage. In this paper we discuss associated query planning and processing aspects. Redshift plays the role of the integration query processor, in addition to the usual processor of queries over Redshift tables. In particular, during query execution, every compute node of a Redshift cluster issues (sub)queries over S3 objects, employing a novel multi-tenant (sub)query execution layer, called Amazon Redshift Spectrum, and merges/joins the results in an streaming and parallel fashion. The Spectrum layer offers massive scalability, with independent scaling of storage and computation. Redshifts optimizer determines how to minimize the amount of data scanned by Spectrum, the amount of data communicated to Redshift and the number of Spectrum nodes to be used. In particular, Redshifts query processor dynamically prunes partitions and pushes subqueries to Spectrum, recognizing which objects are relevant and restricting the subqueries to a subset of SQL that is amenable to Spectrums massively scalable processing. Furthermore, Redshift employs novel dynamic optimization techniques in order to formulate the subqueries. One such technique is a variant of semijoin reduction, which is combined in Redshift with join-aggregation query rewritings. Other optimizations and rewritings relate to the memory footprint of query processing in Spectrum, and the SQL functionality that it is being supported by the Spectrum layer. The users of Redshift use the same SQL syntax to access scalar Redshift and external tables.

1 Introduction and Background

The database literature has described mediators (also named polystores) [6, 1, 4, 2, 3, 5] as systems that provide integrated access to multiple data sources, which are not only databases. In response to a client query that refers to multiple sources, the mediator formulates a distributed query plan, which obtains source data by issuing subqueries and/or (generally) data requests to the sources and consequently merges the results received from the multiple sources.

The advent of scalable object storage systems, such as Amazon's S3, has created a new kind of non-DBMS data source. The S3 data are stored across multiple storage nodes, which are accessible by the multiple *Compute*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

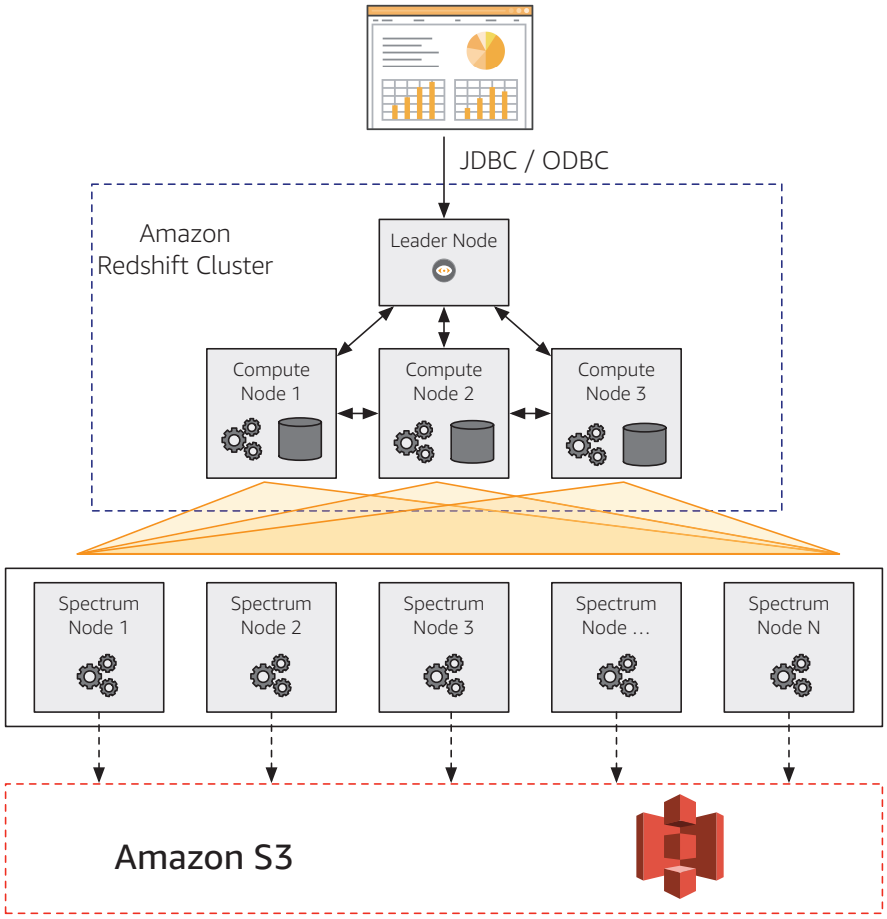

Figure 1: Amazon Redshift and the Spectrum processing layer

Nodes of a parallel database. Many users have data in an Amazon Redshift database and also have data in Amazon S3. A typical use case is very large fact data (in the data cube terminology) residing in S3, with matching dimension tables residing in Amazon Redshift. For such users, Amazon Redshift acts as mediator: It provides a logical view of the S3 data as *external tables* in addition to providing access to the Redshift tables. Queries received by Redshift may refer to both the Redshift tables and the S3 data, while the SQL syntax used when accessing scalar tables, regardless whether they are in Redshift or external, it remains the same. We discuss query planning and query processing issues solved by Redshift.

Using Spectrum in Query Processing. Spectrum is a multi-tenant execution layer of Amazon Redshift that provides massively scalable access and SQL processing capabilities over collections of Amazon S3 objects. It efficiently reads records from objects of various file formats stored in Amazon S3[1] and processes them before streaming the results to Redshift's Compute Nodes. The benefit of massive parallelism is realized when each Spectrum node computes a (sub)query that processes a lot of data but the results returned to Redshift are relatively small. Queries that filter, project and aggregate have this property. Thus, when accessing external tables on S3, Redshift's query optimizer pushes filters, projections and aggregations to the Spectrum layer, while joins (either between local and external tables or even between external tables), order-by's and final aggregations (of

[1]Currently, the supported file formats include columnar formats like Parquet, RCFile and ORC, as well as row-wise formats like CSV, TSV, Regex, Grok, Avro, Sequence, JSON and Ion.

the results of the multiple subqueries that have been sent to the Spectrum layer) are computed at the Redshift Compute Nodes. As we exhibit in Section 2, external table data filtering includes filters that correspond to semi-join reductions (Section 3.4). Furthermore, Redshift is aware (via catalog information) of the partitioning of an external table across collections of S3 objects. It utilizes the partitioning information to avoid issuing queries on irrelevant objects and it may even combine semijoin reduction with partitioning in order to issue the relevant (sub)query to each object (see Section 3.5).

Spectrum's massive parallelism comes with some challenges for the mediator/query planner: The Spectrum nodes are *stateless* - they are not supposed to cooperate and keep state in multi-phase plans. Second, the Spectrum subqueries must be computable within a memory budget; no local disk is used by Spectrum nodes. The hard memory limit leads to the use of partial aggregation (Section 3.3) and run-time decided semijoin reduction (Section 3.4). Further, the SQL functionality supported by each Spectrum node, is not fully aligned to the one of Redshift. Thus, the mediator (Redshift) also considers which SQL processing can be pushed to the Spectrum execution layer and which needs to be complemented in the Redshift processing layer.

2 Running Example

The setting of the scenario is an imaginary future where JK Rowling is launching the 8th book in the original Harry Potter series and the publisher's marketers want to direct the marketing campaign for this book. The particular marketers are in Miami and are looking to build a billboard advertising campaign. They had set such billboard campaigns in the past and they want to find out where and how much they succeeded. So, they want to find out the regions (postal codes) in Miami where the book sales were most improved by the past campaigns. The described scenario was exhibited in Amazon's reInvent 2017 conference, with an exabyte sales (fact) table.[2]

The first step to answer this question is to create a temporary table `hp_book_data` that holds the raw aggregated data about each Harry Potter book sales per Miami zip code, for the sales that followed within 7 days of its release. The second step of the analysis is to compare the book-over-book improvements per zip code and join with knowledge of which release/zip code combinations had billboard campaigns. Thus a comparison can be made between the cases that had the benefit of a campaign and those that did not. The second step is not challenging from a performance point of view, since `hp_book_data` will be very small. Rather the challenge is in computing `hp_book_data`.

The marketers have data in both the Redshift database and on S3. In particular, they have an exabyte fact table `S3.D_CUSTOMER_ORDER_ITEM_DETAILS` stored as a collection of S3 object, while the dimension tables are in Redshift. Having fact tables as external tables on S3 and dimension tables at Redshift table is indeed a common case of Redshift usage of external tables. In this case there are two tables, named `ASIN_ATTRIBUTES` and `PRODUCTS`, that provide information (author, title, release date) about the book editions, which are identified by the `ASIN` numbers. The `REGION` dimension table provides the country, state and city of the `REGION_ID`s that appear in the fact table.

The S3-residing fact table is partitioned by the `ORDER_DAY` attribute. The `partitions` table in the catalog provides the association between `ORDER_DAY`'s and the id's of the object prefixes (partitions) that are associated to each `ORDER_DAY`.

The following query computes the `hp_book_data`.

```
SELECT
   SUM(D.QUANTITY * D.OUR_PRICE) AS SALES,
   P.TITLE, R.POSTAL_CODE, P.RELEASE_DATE
FROM
  S3.D_CUSTOMER_ORDER_ITEM_DETAILS D,
  ASIN_ATTRIBUTES A, PRODUCTS P, REGIONS R
```

[2]The scenario is imaginary and, to the best of our knowledge, does not correspond to a particular marketing campaign.

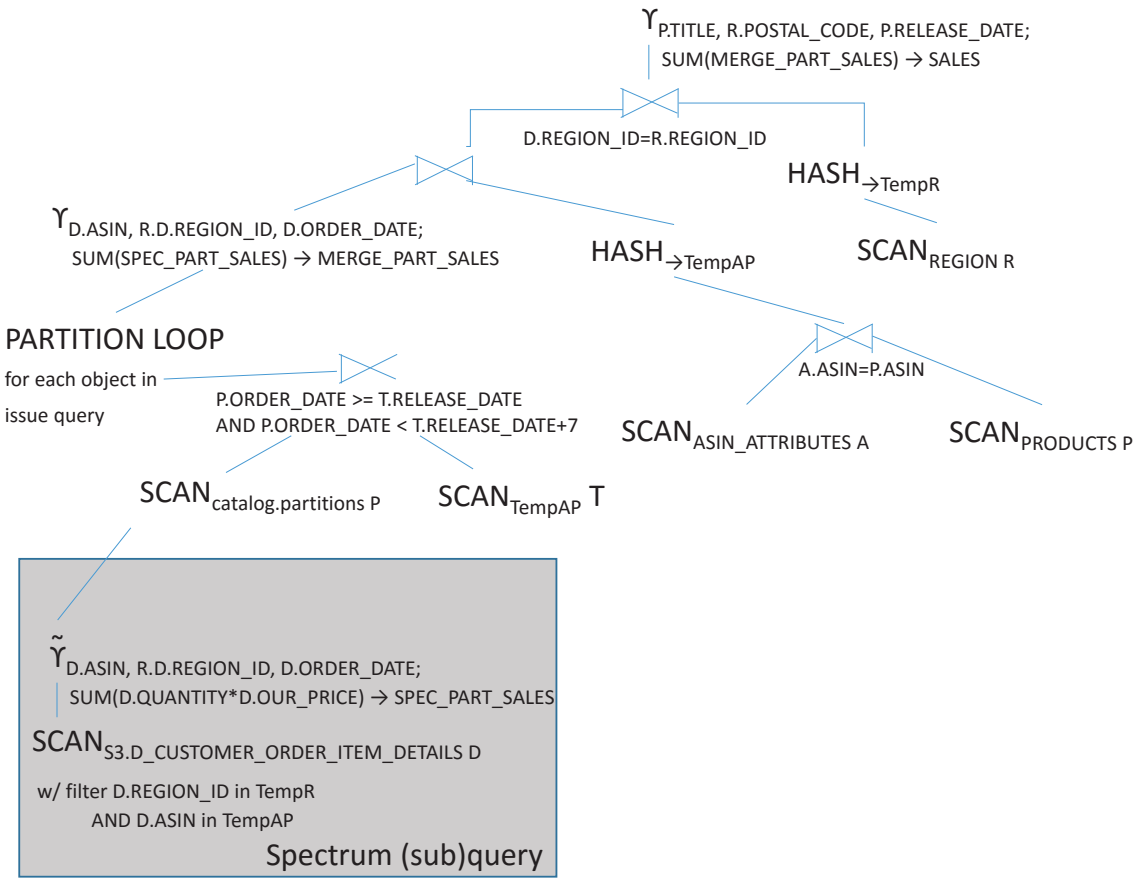

Figure 2: The query plan of the running example. The aggregation operator $\gamma_{\overline{G};f(A)\mapsto S}$ stands for a grouping on the grouping list $\overline{G}$ and aggregating with the aggregate funftion $f(A)$, leading to an aggregate attribute S. The partial aggregation variation $\tilde{\gamma}$ is explained in Section 3.3

```
WHERE
    D.ASIN = P.ASIN AND P.ASIN = A.ASIN AND
    D.REGION_ID = R.REGION_ID AND
    A.EDITION like '%FIRST%' AND P.TITLE like '%Potter%' AND
    P.AUTHOR = 'JK Rowling' AND
    D.ORDER_DAY >= P.RELEASE_DATE AND
    D.ORDER_DAY < P.RELEASE_DATE + 7 Days AND
    R.COUNTRY_CODE='US' AND R.STATE = 'WA' AND
    R.CITY = 'Miami'
GROUP BY
   P.TITLE, R.POSTAL_CODE, P.RELEASE_DATE
```

The plan of Figure 2 executes the query. Informally, its steps are:

1. Combine the Redshift tables `Products` and `ASIN_Attributes` to find the `ASIN`'s, `TITLE`'s and `RELEASE_DATE`'s of the first editions of Harry Potter books. There will be relatively few such tuples.

2. Scan the `REGIONS` table to find the `REGION_ID`'s and `POSTAL_CODE`'s of the Miami regions. Again,

there will be relatively few such tuples.

3. Retrieve from the catalog the list of partitions, identified by ORDER_DAY, that have sales data for the 7 days that followed each release. (See the semijoin in Figure 2.)

4. To each S3 object contained in a qualified partition, issue a Spectrum query that aggregates the revenue per Miami region, book edition and date. Notice the Spectrum query also includes IN filters, sof that only Miami data and Harry Potter book editions are aggregated.

5. On the Redshift side, merge the partially aggregated results per S3 object returned by the Spectrum layer.

6. For each group pull the corresponding values from the dimension tables, by performing the joins.

7. Perform the final aggregation.

The efficiency of the plan comes from Steps 3 and 4. In Step 3, Redshift prunes work at the partition level, and ends up having to only process the (relatively few) objects that follow the 7-days-post-release condition. Consequently, it sends much fewer (sub)queries to the Spectrum layer than a blind querying of all objects would lead to. In Step 4, the large amount of data per object boils down to returning only a few tuples, thanks to the IN filters and the presence of the aggregation. Of course, it matters that the Redshift optimizer first executed the joins of Steps 1 and 2, figures the relevant regions and book editions, and thus focused Steps 3 and Steps 4 to partitions and data in S3 objects that matter.

3 Redshift Dynamic Distributed Query Optimization

We discuss next the optimization steps that Redshift engages into, focusing primarily on special aspects of the optimization.

3.1 Join Ordering

In its first step, the Redshift query optimization creates a query plan, as it would have done even if the S3 table (or S3 tables in the general case) were database tables. In a cost-based fashion, using the statistics of the local and (external) S3 tables it creates the join order that yields the smallest intermediate results and minimizes the amount of data that are exchanged. Since the S3 tables are expected to be much larger than the Redshift tables, the typical plan will have the S3 table(s) at the left-most side of joins sequences. We will focus the rest of our optimization discussion to this pattern.

While join ordering is an expected functionality of databases, it is interesting to note that many SQL-on-Hadoop engines do not offer such. Often their SQL queries are not declarative but rather the join orders have an operational/execution order meaning as well.

3.2 Aggregation PushDown

Pushing the aggregations down to Spectrum has a large effect on performance and scalability, as it dramatically reduces the amount of data each Spectrum node needs to return to the Redshift cluster. In the running example, the aggregation pushdown rewriting leads to Spectrum queries that perform a *pre-aggregation* by grouping on ASIN and REGION_ID. [3] The pre-aggregation of Spectrum, leading to SPEC_PART_SALES is followed by a

[3] From a technical perspective, the ORDER_DATE is also included in the grouping attributes of the pre-aggregation. However, since the example's partitioning dictates that each S3 object has the same ORDER_DATE, the ORDER_DATE is not really accomplishing grouping and could also be eliminated.

merge aggregation that aggregates the Spectrum derived SPEC_PART_SALES into the MERGE_PART_SALES.[4] A final aggregation happens after the joins, since the grouping attributes of the pushed-down aggregation and the query's aggregation are not the same. The generalization of the rewriting behind the push-down of aggregation below a sequence of joins is:[5]

$$\gamma_{\overline{F.G},\overline{D_1.G},\ldots,\overline{D_n.G};agg(e(\overline{F.A}))\mapsto R}(\ldots(F \bowtie_{c_1} D_1)\ldots\bowtie_{c_n} D_n)$$
$$= \gamma_{\overline{F.G},\overline{D_1.G},\ldots,\overline{D_n.G};postagg(preR)\mapsto R}(\ldots(\gamma_{\overline{F.G},\overline{F/c_1\ldots c_n};preagg(e(\overline{F.A}))\mapsto preR}F) \bowtie_{c_1} D_1)\ldots\bowtie_{c_n} D_n)$$

The notation $\overline{Rel.Attr}$ stands for a list of attributes from the table Rel. In the running example, the "fact" table F is the S3 table, which joins twice with dimensions tables (or expressions involving join tables) before the aggregation, i.e., $n = 2$. Figure 2 shows the plan before the aggregation pushdown. The condition c_1 of the first join is `D.ASIN = P.ASIN AND D.ORDER_DAY >= P.RELEASE_DATE AND D.ORDER_DAY < P.RELEASE_DATE + 7 Days` and the condition c_2 of the second join is `D.REGION_ID = R.REGION_ID`. The operator $\gamma_{\overline{F.G},\overline{D_1.G},\ldots,\overline{D_n.G};agg(e(\overline{F.A}))\mapsto R}$ denotes grouping the input on the (concatenation of the) lists of columns $\overline{F.G}$ from the table F and columns $\overline{D_i.G}$ from the dimensions tables D_i. In the running example, $\overline{F.G}$ happens to be empty, i.e., the original plan has no aggregation on the S3 table attributes. The notation $\overline{F/c_1,\ldots,c_n}$ stands for the list of F attributes that appeared in the conditions $c_1,\ldots,c_n$. Informally, the rewriting says that a pre-aggregation can be pushed to F by replacing the dimension attributes in the grouping list with the fact table attributes that appeared in the joins. Notice that despite our use of the intuitive terms "fact table" and "dimension table", the rewriting does not pose foreign key/primary key constraints between the "facts" and "dimensions".

The rewriting requires that the aggregation agg is an associative aggregation, such as SUM, MIN, MAX, etc. The expression e is arbitrary. For each aggregation agg, there is a corresponding pre-aggregation $preagg$ and post-aggregation $postagg$. For example, if the aggregation is $\texttt{COUNT}(e(\ldots)) \mapsto R$ then the preaggregation is $\texttt{COUNT}(e(\ldots)) \mapsto preR$ and the post-aggregation is $\texttt{SUM}(preR) \mapsto R$. Similar pre-aggregation and post-aggregation applies to all associative aggregation operators, such as MIN, MAX, etc. Other aggregate functions such as AVG and STDEV can be emulated by combinations of the associative aggregations. As is well known, a few aggregation operators, such as the median, are neither associative, nor emulatable by associative ones.

There are many special cases that can lead to simpler, more effective rewritings. For example, in certain cases knowledge of the foreign key constraint allows elimination of the post-aggregation. In other cases, the semijoin reduction may render the join at Redshift unnecessary.

Before we proceed in the next optimizations, we raise the question of the rewriting's effectiveness and applicability in the case where the memory needed for grouping exceeds the available main memory.

[4]The merge aggregation is actually two aggregations: A first *local* aggregation by the computing nodes of Redshift and a second, *global* integration of the results of the local one. Depending on the expected cardinality of the aggregation, the global integration step may be executed by a single or more compute nodes. We do not depict this two-staging in Figure 2.

[5]Note, an alternate form of this rewriting would involve an aggregation and a single join, at a time. This would push an aggregation below the join and thus positioning it above the next join in the chain, which would trigger the next application of pushing aggregation below join. In the running example, under this variant we would have (a) a first aggregation at Spectrum on fact table attributes ASIN, ORDER_DAY and REGION_ID, as is the case in Figure 2 also, then (b) the join with the ASIN_ATTRIBUTES and PRODUCTS tables, again as is in Figure 2, (c) an aggregation on dimension table attributes TITLE, RELEASE_DATE and (still) fact table attribute REGION_ID (this is the point where the variant is different from Figure 2), (d) the join with the REGIONS table and (e) the final aggregation on the dimension attributes TITLE, POSTAL_CODE and RELEASE_DATE dictated by the query. At present time we elect the "single" rewriting option that detects the full tree of joins and pushes a single aggregation below it, as typically the very first aggregation (which happens on the Spectrum layer) is responsible for (vastly) most of the performance optimization.

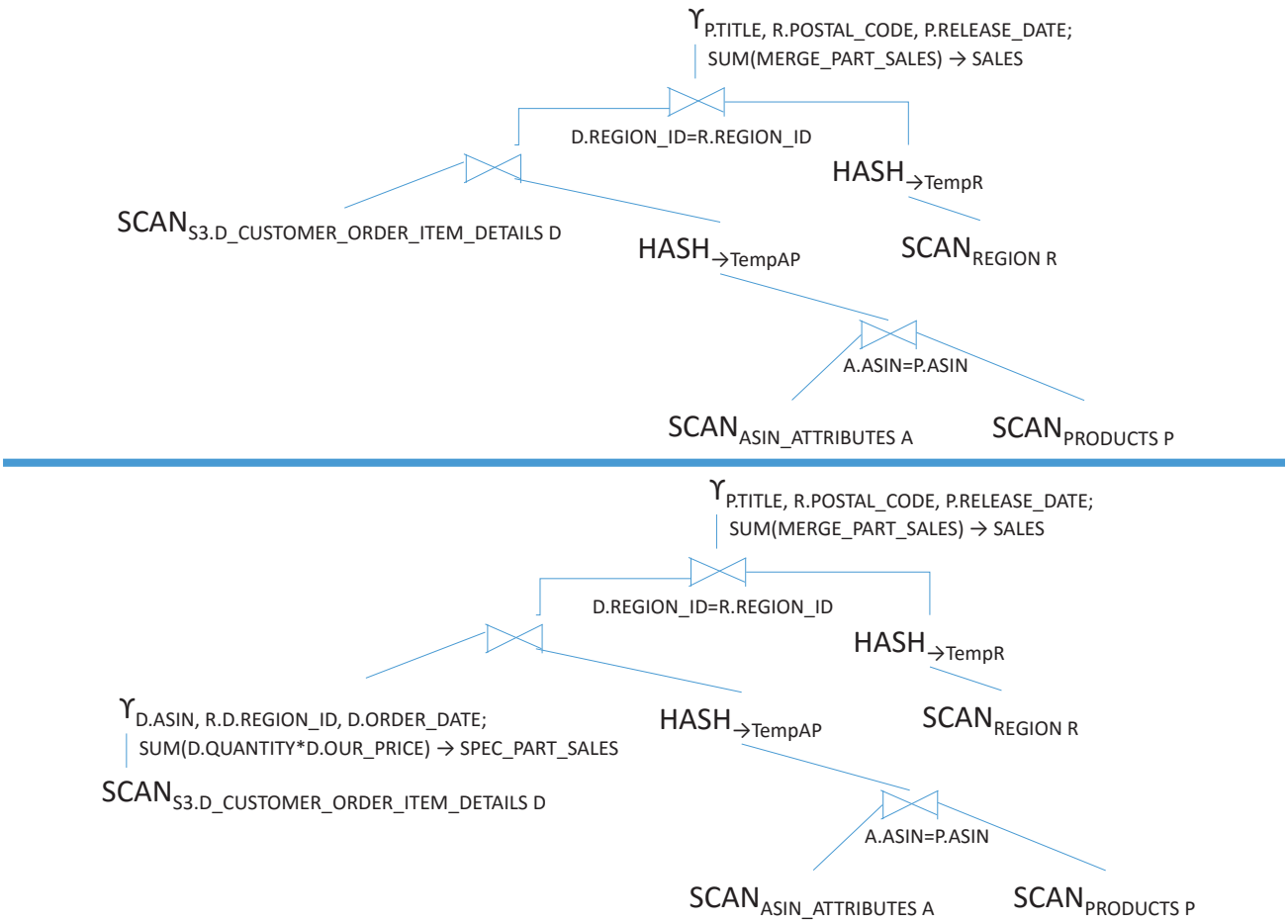

Figure 3: Plans before and after pushing aggregation below joins

3.3 Partial Aggregation

Aggregation pushdown comes with a risk: Aggregation, as usual, employs a data structure with pairs of group-by attribute values and aggregates. For each input tuple, the aggregation finds the relevant pair and updates the aggregate. If the group-by attributes of the input tuple have not appeared in any pair, then a new pair is created. But what if the grouping data structure exceeds the amount of memory that a Spectrum (sub)query may use? In the running example, this can happen if the amount of memory needed to store the pairs consisting of the group-by attributes `ASIN` and `REGION_ID` and the respective aggregate exceed the amount of available memory. This risk is mitigated by turning the pre-aggregation γ into a partial aggregation $\tilde{\gamma}$. A partial aggregation has generally a non-deterministic result and is allowed to output more than one tuples with the same grouping values. This allowance enables $\tilde{\gamma}$ to deal with aggregations that have results larger than the available memory, as follows. Assume that, at some point during the execution of $\tilde{\gamma}$ (a) the grouping structure has reached its maximum memory size and (b) a new input tuple arrives, with a `ASIN` a, `REGION_ID` r and $\langle a, r\rangle$ are not in the grouping structure. The $\tilde{\gamma}$ will pick a pair $(\langle a', r'\rangle, c')$ from the grouping structure, output it, and use its spot in the grouping table for the new entry $(\langle a, r\rangle, 1)$.[6] Multiple replacement strategies are possible. A good strategy

[6]Theoretically, $\tilde{\gamma}$ could output $(a, r, 1)$ and leave the grouping structure as-is. This would probably be the worst replacement strategy, since there is most probably locality in the arrival of tuples with the same `ASIN` and `REGION_ID`.

minimizes the probability that the newly evicted entry will need to be re-established in the grouping table, while keeping low the computation cost of discovering which entry to evict.

3.4 Semijoin Reduction by Dynamic Optimization

In the running example there are multiple regions but relatively few Miami regions. Similarly, there are multiple books but few Harry Potter books. Thus the Spectrum queries should focus on the few Miami regions and Harry Potter books. To achieve this, Redshift engages into semijoin reduction. Once the `REGION_ID`'s and `ASIN`'s are known (i.e., upon having evaluated the respective parts of the join tree), Redshift introduces the discovered `REGION_ID`'s and `ASIN`'s in the Spectrum queries, as `IN` *semijoin filter lists*. Thus a Spectrum query may look like (when formatted in SQL)

```
SELECT REGION_ID, ASIN, SUM(OUR_PRICE*QUANTITY) AS PART_SALES
FROM S3object
WHERE REGION_ID IN [45, 12, 179] AND ASIN IN [35, 6, 17]
GROUP BY REGION_ID, ASIN
```

There is a risk that the semijoin filter lists may be too long and exceed the available memory. Thus the semijoin reduction cannot be absolutely decided during query planning time. Rather, the Redshift query executor makes the decision at run time upon executing the filters and the joins between the dimension tables of Figure 2. If the right hand side results (`TempAP` and `TempR`) turn out to be small enough, then the queries issued to Spectrum will incude the corresponding semijoin filters.

3.5 Smart Partitioning, driven by Joins

The *partition loop* operator is responsible for finding the partitions that contain objects that are relevant to the query and emitting Spectrum queries to them. Its operation is based on the `catalog.partitions` table, which associates each partition of an S3 table with a value of the partition attribute(s) of the S3 table - the `ORDER_DAY` in the example. The partition attribute, which is typically time-related, is often constrained by the query. The simple form of constraining is when a selection filter applies on the partition attribute. For example, this would be the case if the running query also had a condition `D.ORDER_DATE > '04/01/2005'`. Any filter on the partitioned attribute should turn into a filter that is used to find the relevant partitions.

A more interesting case emerges when the constraint on the partition attribute is expressed by a join. In the running query, the `ORDER_DAY`s are constrained by the conditions placed on `PRODUCTS` and `ASIN_ATTRIBUTES`: The only `ORDER_DAY`s that matter are the seven days that followed a Harry Potter book release. Technically, these are the `ORDER_DAY`s in the result `TempAP` in Figure 2. Thus, only the partitions that are associated with these `ORDER_DAY`s should be queried. The *partition loop operator* (Figure 2) finds the partition id's by executing the depicted semijoin of the `catalog.partitions` with `TempAP`. Generally, the semijoin's condition is derived as follows: Detect the join conditions on the partitioned S3 table that involve the partition attribute. Assuming the join condition is in conjunctive normal form, pick the maximum number of conjunction arguments that involve the partition attribute and adjust the partition attribute references to refer to the catalog table (as opposed to the partitioned table).

4 Conclusions

Amazon Redshift provides integrated access to relational tables and S3 objects. The S3 data are accessed via a highly parallel, multi-tenant processing layer, called Amazon Redshift Spectrum. Multiple optimizations ensure that the queries executed at the scalable Spectrum layer process only the relevant S3 objects and return to the Redshift cluster small results, while cost-based optimizations, such as join ordering, are still in effect.

References

[1] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible glue: Scalable self-tunning multi-stores. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[2] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *Proceedings RIDE-DOM*, pages 124–131, 1995.

[3] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, 1994.

[4] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[5] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: souping up big data query processing with a multistore system. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1591–1602, 2014.

[6] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.