OpenIVM: a SQL-to-SQL Compiler for Incremental Computations

Ilaria Battiston
ilaria@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

Kriti Kathuria
kriti.kathuria@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Peter Boncz
boncz@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

ABSTRACT

This demonstration presents a new Open Source SQL-to-SQL compiler for Incremental View Maintenance (IVM). While previous systems, such as DBToaster, implemented computational functionality for IVM in a separate system, the core principle of OpenIVM is to make use of existing SQL query processing engines and perform all IVM computations via SQL. This approach enables the integration of IVM in these systems without code duplication. Also, it eases its use in *cross-system* IVM, i. e. to orchestrate an HTAP system in which one (OLTP) DBMS provides insertions/updates/deletes (deltas), which are propagated using SQL into another (OLAP) DBMS, hosting materialized views. Our system compiles view definitions into SQL to eventually propagate deltas into the table that materializes the view, following the principles of DBSP.

Under the hood, OpenIVM uses the DuckDB library to compile (parse, transform, optimize) the materialized view maintenance logic. We demonstrate OpenIVM in action (i) as the core of a DuckDB extension module that adds IVM functionality to it and (ii) powering cross-system IVM for HTAP, with PostgreSQL handling updates on base tables and DuckDB hosting materialized views on these.

CCS CONCEPTS

• **Information systems** → **Database query processing**.

KEYWORDS

Incremental view maintenance, HTAP, compiler

ACM Reference Format:
Ilaria Battiston, Kriti Kathuria, and Peter Boncz. 2024. OpenIVM: a SQL-to-SQL Compiler for Incremental Computations. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626246.3654743

1 INTRODUCTION & RELATED WORK

Incremental View Maintenance (IVM) is the process of incrementally propagating changes in base tables into a previously computed query result instead of completely re-calculating the query from the changed base tables.

Previous work. IVM has been studied in depth in the past decades, and we consider DBSP [4] the most comprehensive (and recent) work on the matter. Its core idea is to consider each query operator to be performing a stream of integrations and differentiations. Using this principle, it then becomes possible to support arbitrarily complex queries through composition. However, the paper that describes DBSP is conceptual and does not describe a system or strategy that implements its ideas or (consequently) any experimental performance evaluation.

The past decades have also shown that there are different approaches regarding which intermediate results to materialize and maintain. Early strategies were conservative and did not maintain any intermediate results beyond the final materialized result table, limiting the classes of queries that could be maintained efficiently. The DBToaster [1] approach showed that an *aggressive* materialization strategy could provide significantly better view maintenance performance than all techniques preceding it. One should note that efficient IVM still involves trade-offs, not only because creating additional intermediates increases space overhead. The workload also matters, namely the desired granularity and frequency of update propagation, as well as the frequency and cost of queries to the view. Batching changes together, for example, can amortize part of this cost but comes at the price of reduced recency. Ideally, an IVM system should be able to apply different materialization strategies rather than a single one.

On the implementation side, many database systems offer some form of IVM; however, unlike DBSP, they typically have limitations on the complexity of the queries that can be supported. Recent examples include the Snowflake incremental processing engine [2] or Databricks Materialized Views[1]. Notably, both mentioned systems also assume some form of streaming: explicit SQL support for capturing updates, with Databricks specifically advocating using materialized views fed by streams of incoming tuples.

OpenIVM. One motivation for OpenIVM is to construct pipelines that capture streams of updates in one system and feed into materialized views in another system. We call this "*cross-system*" IVM.

Our use case encompasses decentralized privacy-preserving systems: information from personal data stores flows into centralized views, while preserving privacy constraints by guaranteeing coarse-grained aggregation of sensitive attributes.[2]. As another example, an HTAP pipeline could be constructed with cross-system IVM by capturing deltas in an OLTP system and feeding these into an IVM computation that maintains materialized views in an OLAP system.

Previous IVM implementations, such as DBToaster [1] and Materialize[3], opted to create separate computational engines. If the goal is to deploy the IVM into existing database systems, a separate engine *duplicates* query processing functionality. Following the DBSP approach, all incremental computations can be performed using

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3654743

[1] https://www.databricks.com/blog/introducing-materialized-views-and-streaming-tables-databricks-sql
[2] The RDDA [3] (Responsible Decentralized Data Architectures) project
[3] https://materialize.com/

relational ingredients: tables, index structures, algebraic operators, and INSERT/DELETE/UPDATE/UPSERT actions. We, therefore, created OpenIVM as a *compiler*. Our implementation takes in input a database schema and view definition, and generates from there the DDL to create delta tables, possibly intermediate tables and index structures, plus SQL statements that eventually propagate updates from delta tables into the materialized view table. The motivation for our work on SQL-to-SQL IVM is, consequently, to provide a *portable* framework that can be integrated into multiple database systems without adding overhead and duplicating functionality. This utility applies either in a single system, introducing or enhancing its IVM capability, or as a bridge between two systems (cross-system IVM).

We realize that, although PostgreSQL and DuckDB use the same parser, different database systems use different SQL dialects. To ensure portability, we extend our implementation with an additional intermediate tree representation of the operators before emitting SQL, following the technique proposed in the Coral system[4]. Our approach transforms a DuckDB logical plan into a simpler abstract tree (DuckAST), which is then rewritten to a string in the desired SQL dialect, chosen through a flag[5].

A final motivation behind OpenIVM is to provide a tool for IVM research for others to build on, allowing to experiment more easily with different intermediate result materialization strategies and other query optimizations (indexes, relational operators).

In our demonstration of OpenIVM, we show two use-cases: (i) we create a DuckDB [5] extension module that introduces IVM to DuckDB, powered by OpenIVM, and (ii) we show cross-system IVM with changes in PostgreSQL base tables being propagated to materialized views in DuckDB.

2 APPROACH

DBSP Framework. The IVM problem statement is as follows: given a base table T and a view V defined by a query Q over T like $V = Q(T)$, maintain the contents of V as a function of the changes to the base table. More precisely: let ΔT represent the change to the base table T and let ΔV the change to the view V, as a result of executing Q on the new table. Then, IVM seeks to find an algorithm $f : f(\Delta T) = \Delta V$.

DBSP presents a framework that allows finding an f for any arbitrary relational query. It presents two operators $\mathcal{D}$ and $\mathcal{I}$. Let T' be the table after applying a transaction (insertion, delete, update) to T, and V' be the new view after rerunning Q on the new table T'. The *differentiation* operator $\mathcal{D}$ generates the ΔT as $T' - T$, and ΔV as $V' - V$. The *integration* operator $\mathcal{I}$ performs the inverse of $\mathcal{D}$, i.e., reconstitutes the changes such as $T + \Delta T = T'$ and $V + \Delta V = V'$. The notion of + and − is defined precisely in the below paragraphs. DBSP presents the incremental forms of the relational operators, and a series of steps to convert a relational query Q into its incremental form using the incremental operators. This incremental query takes as input ΔT and produces as output ΔV. If a view, defined by a query, is $V = \pi(\sigma(T))$, then its incremental form will be $\Delta V = \pi^*(\sigma^*(\Delta T))$, where the * superscript denotes the incremental version of each operator.

[4]https://github.com/linkedin/coral

[5]The authors wish to acknowledge and thank Akshat Jaimini of Thapar Institute of Engineering & Technology for implementing this functionality.

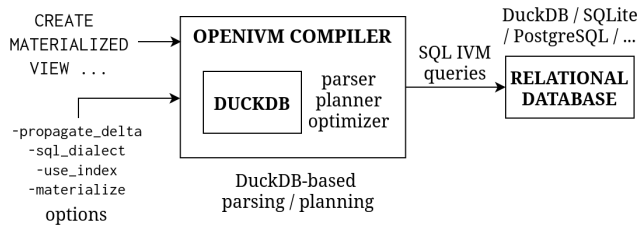

Figure 1: Our IVM engine implementation, consisting of a SQL-to-SQL compiler wrapped around DuckDB. Users can specify the expected optimization strategies through flags.

OpenIVM utilizes the DBSP concepts: first, it generates the logical plan for Q using the DuckDB planner. Then, rewrite rules convert the relational operators to their incremental form. Specifically, the incremental forms of selection and projection operators are the same as their relational form, and the incremental form of a join consists of three relational join operators [4]. The input to the new logical plan is the change to the base table ΔT; the output is ΔV, which is combined with the existing view V.

It is relevant to mention that the DBSP method operates on $\mathbb{Z}$-sets and not the sets consisting in the basis of relational algebra. Thus, all relational constructs defined above must be converted to $\mathbb{Z}$-sets. In order to do so, we associate a weight or *multiplicity* with every element in the set, representing its frequency. For example, the $\mathbb{Z}$-set for the set {*apple*, *banana*} is {*apple* $\longrightarrow$ 1, *banana* $\longrightarrow$ 1}. The + and − in the context of the $\mathcal{D}$ and $\mathcal{I}$ operate on these weights. We use true and false instead of integer weights, representing respectively insertions and deletions in ΔT; these multiplicities carry over to ΔV. In our implementation, tuples with frequency N are modeled with N copies of the same element and multiplicity 1.

To combine ΔV with V, we perform a union for the insertions and a set difference for the deletions, following the semantics of the integration operation $\mathcal{I}$. For aggregate queries such as SUM and COUNT, the boolean multiplicity column itself is insufficient to combine the query result. Therefore, we employ the multiplicity along with the aggregation function output. For example, if the ΔV is {*apple* $\longrightarrow$ (false, 3), *banana* $\longrightarrow$ (true, 1)} and V is {apple $\longrightarrow$ (true, 5), *banana* $\longrightarrow$ (true, 2)}, then the new view V' would be {*apple* $\longrightarrow$ (true, 2), *banana* $\longrightarrow$ (true, 3)}: 3 units of *apple* were removed and 1 unit of *banana* was added.

The Compiler: DuckDB inside OpenIVM. DuckDB [5] is an embedded analytical DBMS originally developed at CWI; it caters to many analytical use cases, including data science and data transformation pipelines, and can be deployed in mobile apps, in-browser (using WASM), as well as laptops and even cloud computing, thanks to its small footprint, portable code, and embeddable nature.

Figure 1 shows how the OpenIVM SQL-to-SQL compiler takes advantage of DuckDB being an embedded database, linking it as a library. This gives it access to the DuckDB SQL parser, planner, and optimizer, which are crucial infrastructures for IVM query rewriting and optimization.

It is not required to fork DuckDB to extend its functionality, as it allows to write *extension* modules in C++. These can be loaded on the fly in a running DuckDB instance to extend its catalog,

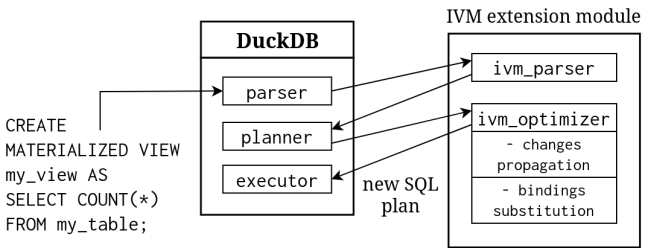

Figure 2: Our IVM optimizer extension, interacting with the DuckDB core engine.

access methods, parser, optimizer, and execution operators. Parsers, however, are notoriously non-extensible. The DuckDB approach here is first to use its own parser, but on syntax errors, try to re-parse a SQL statement with fall-back parsers provided by extensions. An extension module registers its new functionality by calling DuckDB registration functions. These registration functions can also be called directly from an application that uses DuckDB as a library. This is what our SQL-to-SQL compiler does: it registers some extension functionality, namely a parser extension and extra rewrite rules in the optimizer.

Similar to DuckPGQ [6] (which adds SQL/PGQ syntax), we developed a simple fall-back parser that recognizes the `CREATE MATERIALIZED VIEW` syntax, removes the `MATERIALIZED` keyword and finally returns into the original DuckDB parser. The query is then transformed by DuckDB into a tree of parsed statements, then into a logical algebra tree, and subsequently optimized. As a final step in the optimization, DuckDB will call the OpenIVM extension rules. Here, we substitute bindings at the leaves such that the query is executed against the changes rather than the original table. The DBSP-style rewriting of the relational query operators is performed in a bottom-up fashion, rewriting all relational operators into their incremental equivalents.

In terms of query optimization, aggregation, for instance, allows building an index on the materialized aggregation table (using the `GROUP BY` columns as keys). Similarly, one can think of various relational strategies or custom operators to incorporate changes in a materialized aggregation: replacing the materialized table with a `UNION` and regrouping, or through a full-outer-join, or maintaining it with a left-join with an `UPSERT` (ignoring deletions here for simplicity). The materialization strategy (eager, none, or something in between) provides a similar search space for alternative IVM computation plans. For now, we only offer a small number of alternatives, and choosing one is controlled manually using compiler switches; but as we implement join operations, the search space should increase, and *cost-based* optimization should then make these choices, paving the way for new research opportunities.

Another important open question in cross-system IVM is how to propagate changes from T to ΔT. This process could be done in many ways: through triggers, optimizer rules, or not at all in systems such as append-only streams. This choice is also currently left to the user – optimizer rules are being developed for DuckDB; however, for PostgreSQL (or any alternative system), users are required to configure these triggers independently.

The Extension Module: OpenIVM inside DuckDB. While the OpenIVM SQL-to-SQL compiler can be used as a standalone command-line tool, we also wrap it into a dynamically loadable DuckDB extension module. Our implementation, as shown in Figure 2, extends the DuckDB query processing pipeline. The purpose of such, specifically, is to add IVM to DuckDB *natively*. To achieve this, when the fall-back parser parses a `CREATE MATERIALIZED VIEW`, we execute the compiled output to create the delta tables as well as any generated intermediate result tables or indexes, along with a table that represents the materialized result. We store the SQL scripts that propagate the contents of the delta tables to the materialized view table on the disk to allow future inspection and usage without having to start DuckDB. Furthermore, another optimizer rule can then be used to intercept `INSERT`/`DELETE`/`UPDATE` statements into the base tables. These are translated into actions that carry through these base table changes, fill the delta tables ΔT, and kick off the SQL propagation scripts.

One single materialized view V definition corresponds to multiple SQL instructions: the ΔT are created for each T, along with their eventual indexes and boolean multiplicity columns. Furthermore, the $\Delta_i V$ are generated, where $i = 1, \ldots N$ and N is the number of views to materialize. Currently, our IVM implementation only supports single-table ($N = 1$) projections, filters, grouping, and the aggregates `SUM` and `COUNT`; we are in the process of extending it to `MIN`, `MAX` and `JOIN`.

Internally, we store materialized views as tables and save their additional properties – query plan, SQL string, query type – in metadata tables. Once the setup for IVM has been generated, we can process the queries to compute incrementally. This step happens by implicitly calling a table function, adding a dummy node to the plan of the original query to trigger the optimizer rules.

However, more is needed to translate the newly generated logical representation to SQL: IVM requires multiple post-processing steps to ensure consistency of ΔT. These comprehend:

(1) Insertion in ΔV of the tuples resulting from querying ΔT.
(2) Insertion or update in V of the newly-inserted tuples in ΔV, removing the multiplicity column.
(3) Deletion of the invalid rows in V, e. g. the ones with `SUM` or `COUNT` equal to 0, or `false` multiplicity without aggregate.
(4) Deletion from ΔT and ΔV after applying the changes.

The SQL instructions emitted for Step 2 can drastically change depending on the input query. For example, a `GROUP BY` query with an aggregation function can be translated into a `LEFT JOIN` rather than a `UNION`, depending on the complexity of operators.

As we show during the demonstration, preliminary results indicate clear improvements in resource consumption by executing incremental computations rather than running the query against the whole dataset. We also examine the overhead caused by having a materialized index: its creation is necessary for aggregate queries, as DuckDB requires an index to apply upserts. The ART (Adaptive Radix Tree) is generated after having populated V, as it is more efficient to build small indexes for each chunk and merge them. However, its creation only adds significant overhead the first time, and it can be used in the future to speed up joins, especially when the joined part has just a few unique keys.

3 DEMONSTRATION SCENARIOS

Enabling IVM inside DuckDB We demonstrate our system, providing visitors with two scenarios. We set up a GitHub repository containing our OpenIVM extension[6] that implements table functions to run queries and benchmarks. Our demonstration aims to familiarize users with incremental representations of computations in SQL and how easily they can be plugged into relational systems.

We allow visitors to run arbitrary queries on our DuckDB OpenIVM infrastructure. Furthermore, we provide pre-loaded datasets to experiment with and additional ideas for queries to write in the DuckDB shell. Users can then examine the compiled output and check the correct result of the incremental computations applied to V. Here, we show an example of what a query emitted by our compiler looks like and the necessary steps to obtain it.

Listing 1: Example DDL for our IVM setup

```
CREATE TABLE groups(group_index VARCHAR,
    group_value INTEGER);
CREATE MATERIALIZED VIEW query_groups AS SELECT
    group_index, SUM(group_value) AS total_value
    FROM groups GROUP BY group_index;
```

Given this DDL, our IVM compiler will emit new queries (omitted for brevity) to provide the required tables ΔT and ΔV to store changes. Then, the compiler generates the SQL statements to propagate modifications, as shown below.

Listing 2: Generated SQL instructions for IVM

```
INSERT INTO delta_query_groups
SELECT group_index, SUM(group_value) AS
    total_value, _duckdb_ivm_multiplicity
FROM delta_groups
GROUP BY group_index, _duckdb_ivm_multiplicity;
INSERT OR REPLACE INTO query_groups
WITH ivm_cte AS (
SELECT group_index,
    SUM(CASE WHEN _duckdb_ivm_multiplicity = FALSE
         THEN -total_value ELSE total_value END)
        AS total_value
FROM delta_query_groups
GROUP BY group_index)
SELECT query_groups.group_index,
    SUM(COALESCE(query_groups.total_value, 0) +
        delta_query_groups.total_value)
FROM ivm_cte AS delta_query_groups
LEFT JOIN query_groups ON query_groups.group_index
     = delta_query_groups.group_index
GROUP BY query_groups.group_index;
DELETE FROM query_groups WHERE total_value = 0;
DELETE FROM delta_query_groups;
```

These SQL commands can either be run eagerly, i. e. every time a change is registered on the base table, or lazily, i. e. refreshing the materialized view when it is queried. In our examples, we choose to employ the latter approach.

Once we demonstrate the potentialities of our compiler, we want to give users the possibility to benchmark it and test it in DuckDB. We offer different benchmarks with sets of pre-written `GROUP BY` queries to show how computationally intensive each part of the incremental maintenance is. We argue that the incremental computation approach is more efficient than recalculating V each time it is queried.

[6]https://github.com/ila/duckdb/tree/rdda/extension/openivm

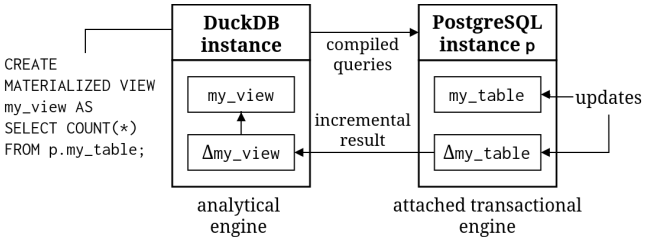

Figure 3: Our cross-system IVM demo, showcasing an HTAP workload with DuckDB and PostgreSQL.

Cross-System IVM The architecture for our cross-system IVM is depicted in Figure 3: we emulate a transactional workload with PostgreSQL using the datasets previously mentioned. We allow users to input analytical queries to be run from DuckDB to an attached PostgreSQL instance. The data stored on PostgreSQL is accessed via the DuckDB integration with PostgreSQL, which gives access to the PostgreSQL schemata and allows cross-system queries. The result of the analytical query is stored in a materialized view in DuckDB. This output is incrementally maintained using the SQL plans generated by our IVM compiler. Visitors can query data stored on PostgreSQL using the DuckDB shell. Furthermore, we provide a script to run the pipeline automatically given an input query Q. The output will be the result of Q on the data on PostgreSQL and its changes, stored in a DuckDB materialized view.

We also allow users to benchmark our system: we show a transparent comparison of the query performance in pure DuckDB, pure PostgreSQL, cross-system, and without IVM.

Using the OpenIVM system, therefore, makes it possible to combine a trusted and efficient OLTP system (PostgreSQL) with an efficient analytical engine (DuckDB) and also easily transform data from operational tables that are maintained using OLTP into warehoused views for OLAP use cases; thus providing a practical HTAP approach.

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. arXiv:1207.0137 [cs.DB]
[2] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, Dan Sotolongo, and Boyuan Zhang. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* 1, 2, Article 196 (jun 2023), 27 pages. https://doi.org/10.1145/3589776
[3] Ilaria Battiston and Peter Boncz. 2023. Improving data minimization through decentralized data architectures.
[4] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. arXiv:2203.16684 [cs.DB]
[5] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 1981–1984.
[6] Daniël ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *Proceedings of the Conference on Innovative Data Systems Research.*