An SQL-to-M3 compiler for Factorized Incremental View Maintenance

Alexandru Văleanu

Oriel College

University of Oxford

Supervisors: Prof. Dan Olteanu, Dr Milos Nikolic

Final Honour School of Computer Science - Part B

Trinity 2018

Abstract

Recently, a new algorithm for solving the Incremental View Maintenance problem has been proposed by Dan Olteanu and Milos Nikolic in [NO18] called Factorized Incremental View Maintenance, or F-IVM for short. F-IVM is a higher-order IVM algorithm that reduces the problem of maintaining an arbitrary query with joins, projections, and group-by aggregates to maintaining a hierarchy of increasingly simpler views. This algorithm may obtain a significantly lower time (and memory) complexity by factorizing the computation of the updates, keys and payloads. This thesis provides an implementation of this algorithm while creating a lightweight full-scale efficient software system which can be easily deployed in order to solve arbitrary instances of the IVM problem.

Acknowledgements

I would like to express my gratitude to my supervisors Prof. Dan Olteanu and Dr Milos Nikolic for our weekly meetings which helped me understand the topic. I am thankful for their aspiring guidance, constructive criticism and friendly advice during the whole project. I also want to thank Dr Milos Nikolic for his invaluable help with DBToaster and everything implementation-related.

To my mother

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modern software applications, ranging from E-commerce to algorithmic trading, require real-time data analytics, which enables the application to make faster, more precise and more effective decisions than conventional decisions made with stale data or no data, based on views over databases that change at very high rates. Such views must be maintained as cheaply and fast as possible. The data management problem of maintaining these views over very large, evolving databases is called the Incremental View Maintenance (IVM) problem. Rudimentary database systems usually take the naive approach of recomputing the whole query every time the database changes. This approach and these systems cannot be used in most real-life applications that usually operate using large quantities of raw data.

My supervisors, Dan Olteanu and Milos Nikolic, have recently introduced F-IVM [NO18], a unified IVM approach for solving a wide variety of tasks. F-IVM uses the fact that updates are usually local (i.e. they only affect a small fraction of the database) so we can efficiently maintain the result of $Q(R + \Delta R)$ by exploiting $Q(R)$ and some delta query $\Delta Q(R, \Delta R)$.

In contrast to classical (first-order) IVM, which does not use extra views and computes changes in the query result on the fly [CY12], F-IVM can solve the maintenance problem much faster and with lower theoretical complexity using carefully chosen extra views. F-IVM may also use substantially fewer and cheaper views than fully-recursive IVM, which is the approach taken by the state of the art IVM system DBToaster [KAK$^+$14].

In this project, we will show how to implement F-IVM as an intuitive, generic, full-scale software system and how to apply it to a couple of scenarios.

1.2 Project goals

The main goal of this project is to take the framework introduced in the recently published research paper [NO18] and turn in into an actual software application. We identify several critical requirements for our system:

- Our system must be able to take as input a SQL query and a variable order and produce the M3 trigger program that is fed into the back-end of DBToaster effectively replacing the front-end component of the system. DBToaster is an open-source SQL-to-native-code compiler that generates embeddable query engines. There are two main reasons for this requirement. Firstly, we would like to have a full solution to the IVM problem and DBToaster's back-end is exactly what we need in order to compile our M3 code into a functional query engine. Secondly, all previously worked out use cases of F-IVM, used to test and demonstrate features of the framework, have been extensions of DBT coded using M3. In *Chapter 6: Experiments*, we compare our system with manually-optimized use cases of F-IVM, as presented in [NO18].
- The system must be flexible enough to support a wide variety of SQL queries; at the very least it must support the handful of queries used throughout the paper [NO18].
- The system must be able to build a graphical representation of its output, which consists of the optimized maintenance strategy, the high-level DBToaster trigger code generated for each view in the maintenance strategy as well as the declarative view definition. This will help the user analyze and better understand both the framework and the system.
- Like any other good software system, ours should be easily maintainable and extensible. In the final chapter of this thesis we will mention some features we discussed but did not have time to fully implement.

1.3 Related work

There exists a multitude of algorithms for tackling the IVM problem implemented in a wide variety of systems (both commercial and open-source). Most commercial databases, e.g., Oracle [Ora] and SQLServer [SQL], support IVM for restricted classes of queries. LogicBlox supports higher-order IVM for Datalog programs [AtCG+15, GOW15]. Trill is a high-performance incremental query processor for diverse analytics but does not support

complex aggregates like cofactor matrices [CG+14]. DBToaster supports fully recursive higher-order IVM [KAK+14] for general queries but it is much slower while using more memory than F-IVM.

However, so far none of these make use of this revolutionary, state of the art approach to factorized IVM, thus making our system the first full-scale implementation of F-IVM. A demonstration paper of the system developed in this project will be submitted later in the year to a tier-1 international conference on database systems.

1.4 Project outline

The thesis is structured as follows:

- In **Chapter 2**, we introduce the theoretical foundations of our framework
- In **Chapter 3**, we give a concise presentation of F-IVM
- In **Chapter 4**, we give a detailed overview of our system's implementation
- In **Chapter 5**, we show how our system can be used to solve some real-life applications
- In **Chapter 6**, we compare the code generated by our system with highly-optimized handwritten code
- In **Chapter 7**, we present the conclusions as well as possible extensions of our system

Chapter 2

Preliminaries

Summary

In this chapter we provide the background material necessary for understanding our framework. This introduction follows the same structure, notations and content as the preliminaries section from [NO18].

2.1 Rings

In this section we define **rings**, algebraic structures which are used throughout the whole thesis and are vital to our framework.

Definition 2.1.1 A **ring** $(\mathcal{I}, +, \times, \mathbf{0}, \mathbf{1})$ is a set $\mathcal{I}$ with binary operations $+$ and $\times$ satisfying the following set of axioms, called **ring axioms**:

- $+$ *is associative:* $(a + b) + c = a + (b + c)\ \forall a, b, c \in \mathcal{I}$
- $+$ *is commutative:* $a + b = b + a\ \forall a, b \in \mathcal{I}$
- $\mathbf{0}$ *is the additive identity:* $a + \mathbf{0} = a\ \forall a \in \mathcal{I}$
- *-a is the additive inverse of a:* $\forall a \in \mathcal{I}\ \exists$ *-a* $\in \mathcal{I}$ *such that* $a - a = \mathbf{0}$
- $\times$ *is associative:* $(a \times b) \times c = a \times (b \times c)\ \forall a, b, c \in \mathcal{I}$
- $\mathbf{1}$ *is the multiplicative identity:* $a \times \mathbf{1} = a\ \forall a \in \mathcal{I}$
- $\times$ *is distributive with respect to* $+$
  - *Left distributivity:* $a \times (b + c) = (a \times b) + (a \times c)\ \forall a, b, c \in \mathcal{I}$
  - *Right distributivity:* $(b + c) \times a = (b \times a) + (c \times a)\ \forall a, b, c \in \mathcal{I}$

Definition 2.1.2 A **semiring** $(\mathcal{I},\ +,\ \times,\ \mathbf{0},\ \mathbf{1})$ satisfies all axioms above except the additive inverse one and also satisfies one extra axiom: $\mathbf{0} \times a = a \times \mathbf{0} = \mathbf{0}$.

Definition 2.1.3 A **ring** (or **semiring**) is commutative if $\times$ is commutative (i.e. $a \times b = b \times a\ \forall a, b \in \mathcal{I}$).

Example 2.1.4 The sets $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ together with arithmetic operations $+$ and $\times$ and numbers 0 and 1 form commutative rings. The set $\mathbb{N}$ of natural numbers doesn't have the additive inverse property so it only forms a commutative semiring.
The set $\mathcal{M}$ of $(n \times n)$ matrices forms a non-commutative ring $(\mathcal{M},\ +,\ \times,\ 0_{n,n},\ I_n)$ where $0_{n,n}$ and I_n are the zero matrix and the identity matrix of order n.
See *Chapter 5: Applications* for a much more exotic ring.

□

2.2 Data model and Query language

2.2.1 Data model

In this section we introduce the notations and definitions used in describing our data.

Definition 2.2.1 A **schema** $\mathcal{S}$ is a set of variables (or attributes).

Definition 2.2.2 The **Domain** of a variable $X \in \mathcal{S}$ is the set of all possible values of X.

Definition 2.2.3 A tuple t over schema $\mathcal{S}$ is a finite ordered list (sequence) of elements where each element is a distinct variable of $\mathcal{S}$. We define the Domain of a tuple t over schema $\mathcal{S}$ as Domain(t) $= \prod_{X \in \mathcal{S}} Domain(X)$.
Note: We use $\langle\rangle$ to represent the empty tuple over the empty schema.

Example 2.2.4 Let $\mathcal{S} = \{X_1, X_2, X_3\}$ with $Domain(X_i) = \{1, 2, 3, 4\}$. $\langle 2, 3, 1\rangle$ is a tuple over $\mathcal{S}$. So is $\langle 1, 1, 1\rangle$. But $\langle 1, 2, 5\rangle$ is not because $5 \notin Domain(X_3)$.
Please note that $\langle\rangle \notin \mathcal{S}$ because $\mathcal{S}$ is not empty.

Definition 2.2.5 Let $(\mathcal{I},\ +,\ \times,\ \mathbf{0},\ \mathbf{1})$ be a ring.
A relation R over schema $\mathcal{S}$ and ring $\mathcal{I}$ is a function $\mathsf{R} : Domain(\mathcal{S}) \to \mathcal{I}$ which maps tuples over schema $\mathcal{S}$ to elements in $\mathcal{I}$ and $\mathsf{R}[t] \neq \mathbf{0}$ for finitely many tuples t $\in \mathcal{S}$.
We call a tuple t a key if and only if $\mathsf{R}[t] \neq \mathbf{0}$. We call $\mathsf{R}[t]$ the payload of t in R.

Example 2.2.6 Let's assume we have a schema $\mathcal{S} = \{A, B\}$ with $Domain(A) = \{a, b\}$ and $Domain(B) = \{\frac{1}{2}, \frac{2}{3}, \frac{3}{4}\}$. Over this schema and ring $(\mathbb{R}, +, \times, 0, 1)$ we can define a relation $\mathsf{R} = \{\langle a, \frac{1}{2}\rangle \to 100, \langle a, \frac{3}{4}\rangle \to 200, \langle b, \frac{1}{2}\rangle \to 120\}$. Notice that although $\langle b, \frac{2}{3}\rangle$ is a valid tuple over $\mathcal{S}$, $\langle b, \frac{2}{3}\rangle \notin R$. In this case, we can say that $\mathsf{R}[\langle b, \frac{2}{3}\rangle] = 0$.

□

Definition 2.2.7 A database $\mathcal{D}$ is a collection of relations over the same ring.

According to [NO18], this data model is in line with prior work on K-relations over provenance semirings [GKT07], generalized multiset relations [Koc10], and factors over semirings [AKNR16].

2.2.2 Query language

We consider SQL queries with SUMs, joins and group-by aggregates of the following type:

```
SELECT X_1, ..., X_k, SUM(f_{k+1}(X_{k+1}) * ... * f_m(X_m))
FROM R_1 NATURAL JOIN ... NATURAL JOIN R_n
GROUP BY X_1, ..., X_k;
```

Definition 2.2.8 The variables which appear in a GROUP BY clause, X_1, ..., X_k, are *free*, while all other variables X_{k+1}, ..., X_m are *bound*.

Definition 2.2.9 The result of R NATURAL JOIN S is the set of all combinations of tuples in the two relations, R and S, that are equal on their common attribute names.

Definition 2.2.10 The functions $f_i : \text{Domain}(X_i) \to \mathcal{I}$ for $k < i \leq m$, also called *lifting functions*, are used to map variable values to elements (payloads) in $\mathcal{I}$.

Note: Notice that the $*$ operator used inside the argument of SUM is the $\times$ operator from the ring and that the $+$ which is used internally by SUM is from the ring too.

Definition 2.2.11 (adapted from [NO18]) Given a ring $(\mathcal{I}, +, \times, \mathbf{0}, \mathbf{1})$, relations R and S over schema $\mathcal{S}_1$ and relation T over schema $\mathcal{S}_2$, a variable $X \in \mathcal{S}_1$, and a lifting function $f_X : Domain(X) \to \mathcal{I}$, we define the operators $\uplus, \otimes$ and $\bigoplus$ as follows:

union:

$$\forall \mathbf{t} \in \mathsf{D}_1\text{:}\ \ (\mathsf{R} \uplus \mathsf{S})[\mathbf{t}] = \mathsf{R}[\mathbf{t}] + \mathsf{S}[\mathbf{t}]$$

join:

$$\forall \mathbf{t} \in \mathsf{D}_2\text{:}\ (\mathsf{S} \otimes \mathsf{T})[\mathbf{t}] = \mathsf{S}[\pi_{\mathcal{S}_1}(\mathbf{t})] * \mathsf{T}[\pi_{\mathcal{S}_2}(\mathbf{t})]$$

aggregation by marginalization:

$$\forall \mathbf{t} \in \mathsf{D}_3\text{:}\ (\textstyle\bigoplus_X \mathsf{R})[\mathbf{t}] = \sum \{\, \mathsf{R}[\mathbf{t}_1] \ \times\ f_X(\pi_{\{X\}}(\mathbf{t}_1)) \mid \mathbf{t}_1 \in \mathsf{D}_1, \mathbf{t} = \pi_{\mathcal{S}_1 \setminus \{X\}}(\mathbf{t}_1)\}$$

where $\mathsf{D}_1 = Domain(\mathcal{S}_1)$, $\mathsf{D}_2 = Domain(\mathcal{S}_1 \cup \mathcal{S}_2)$, $\mathsf{D}_3 = Domain(\mathcal{S}_1 \setminus \{X\})$, and $\pi_{\mathcal{S}}(\mathbf{t})$ is a tuple representing the projection of tuple $\mathbf{t}$ on the schema $\mathcal{S}$.

Using the operators defined above, we can encode our SQL query as:

$$\mathsf{Q}[X_1, \ldots, X_k] = \bigoplus_{X_{k+1}} \cdots \bigoplus_{X_m} \bigotimes_{1 \le i \le n} \mathsf{R_i}[\mathcal{S}_i],$$

where $\otimes$ is the join operator, $\bigoplus_{X_{k+1}}$ is the aggregation operator that marginalizes over the variable X_{k+1}, and each relation $\mathsf{R}_i : \mathit{Domain}(\mathcal{S}_i) \to \mathcal{I}$ maps tuples (keys) defined over schema $\mathcal{S}_i$ to elements (payloads) in $\mathcal{I}$.

Example 2.2.12 Consider the following relations over the ring $(\mathcal{I}, +, \times, \mathbf{0}, \mathbf{1})$:

A	B	$\to \mathsf{R}[A,B]$
a_1	b_1	$\to r_1$
a_2	b_1	$\to r_2$

A	B	$\to \mathsf{S}[A,B]$
a_1	b_1	$\to s_1$
a_2	b_7	$\to s_3$

B	C	$\to \mathsf{T}[B,C]$
b_7	c_2	$\to t_1$
b_1	c_2	$\to t_2$

and the query $\mathsf{Q}[\,] = \bigoplus_C(\bigoplus_B(\bigoplus_A(\mathsf{R} \uplus \mathsf{S}) \otimes T))$.

We can then compute:

A	B	$\to (\mathsf{R} \uplus \mathsf{S})[A,B]$
a_1	b_1	$\to r_1 + s_1$
a_2	b_1	$\to r_2$
a_2	b_7	$\to s_3$

A	B	C	$\to \big((\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T}\big)[A,B,C]$
a_1	b_1	c_2	$\to (r_1 + s_1) \times t_2$
a_2	b_1	c_2	$\to r_2 \times t_2$
a_2	b_7	c_2	$\to s_3 \times t_1$

B	C	$\to \big(\bigoplus_A(\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T}\big)[B,C]$
b_1	c_2	$\to (r_1 + s_1) \times t_2 \times f_A(a_1) + r_2 \times t_2 \times f_A(a_2)$
b_7	c_2	$\to s_3 \times t_1 \times f_A(a_2)$

C	$\to \big(\bigoplus_B(\bigoplus_A(\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T})\big)[C]$
c_2	$\to ((r_1 + s_1) \times t_2 \times f_A(a_1) + r_2 \times t_2 \times f_A(a_2)) \times f_B(b_1) + (s_3 \times t_1 \times f_A(a_2)) \times f_B(b_7)$

$\langle\rangle$	$\to \big(\bigoplus_C(\bigoplus_B(\bigoplus_A(\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T}))\big)[\,]$
$\langle\rangle$	$\to (((r_1 + s_1) \times t_2 \times f_A(a_1) + r_2 \times t_2 \times f_A(a_2)) \times f_B(b_1) + (s_3 \times t_1 \times f_A(a_2)) \times f_B(b_7)) \times f_C(c_2)$

where the values r_1, r_2, s_1, s_3, t_1, t_2 are non-$\mathbf{0}$ values from $\mathcal{I}$ and $f_A : \mathit{Domain}(A) \to \mathcal{I}$, $f_B : \mathit{Domain}(B) \to \mathcal{I}$ and $f_C : \mathit{Domain}(C) \to \mathcal{I}$ are the lifting functions for A, B and C.

Notice that Q is also a relation (over the empty schema) that maps the empty tuple (key) to the result of the query (payload in $\mathcal{I}$).

□

Example 2.2.13 Let us consider the following SQL query over tables $R(A, B)$ and $S(B, C)$:

```
SELECT SUM(1)
FROM R NATURAL JOIN S;
```

Assuming we use $\mathbb{Z}$ as our ring, we can encode the table R as a relation $\mathsf{R} : Domain(A) \times Domain(B) \to \mathbb{Z}$ that maps tuples $\langle a, b \rangle$ to their multiplicity in R. Similarly, we encode table S as relation $\mathsf{S} : Domain(B) \times Domain(C) \to \mathbb{Z}$.
Our SQL query is then translated into:

$$\mathsf{Q}[\,] = \bigoplus_A \bigoplus_B \bigoplus_C (\mathsf{R}[A, B] \otimes \mathsf{S}[B, C])$$

where the lifting functions f_A, f_B and f_C used in any $\bigoplus$ map all tuples to 1.
The relation Q maps the empty tuple $\langle\rangle$ to the count.

□

Example 2.2.14 In this example, we will show how to encode into our formalism a much more complex SQL query. Let us consider the following query over tables $R(A, B, C)$, $S(B, D, E)$ and $T(D, E, F)$:

```
SELECT R.A, S.B, T.F, SUM(R.C * S.D + S.E * T.E)
FROM R NATURAL JOIN S NATURAL JOIN T
GROUP BY R.A, S.B, T.F;
```

For simplicity, let us assume that C, D and E all take values from $\mathbb{Z}$ (i.e. $Domain(C) = Domain(D) = Domain(E) = \mathbb{Z}$). We encode the tables R, S, and T as relations mapping tuples to their multiplicity, as we did in the previous example. Variables A, B, F are free in Q while C, D and E are bound. Each lifting function maps the value of a variable to itself, so we have $\forall x \in \mathbb{Z} : f_C(x) = f_D(x) = f_E(x) = x$. The SQL query can be encoded as:

$$\mathsf{Q}[A, B, F] = \bigoplus_C \bigoplus_D \bigoplus_E (\mathsf{R}[A, B, C] \otimes \mathsf{S}[B, D, E] \otimes \mathsf{T}[D, E, F])$$

The computation of the aggregate `SUM(R.C * S.D + S.E * T.E)` now happens in the ring $\mathcal{I}$, over payloads.

□

2.3 Variable orders

Definition 2.3.1 A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.

All modern DBMSs include a component called query optimizer, which evaluates some of the different, correct possible plans for executing the query and returns what it considers the best option.

Listing 2.1: Example of a traditional query plan

```
StmtText
----
  |--Sort(ORDER BY:([c].[LastName] ASC))
       |--Nested Loops(Inner Join, OUTER REFERENCES:([e].[C], [Expr1004]) WITH
           UNORDERED PREFETCH)
            |--Clustered Index Scan(OBJECT:([A].[H].[E].[PK_E_EID] AS [e]))
            |--Clustered Index
                Seek(OBJECT:([A].[Person].[Contact].[PK_Contact_C] AS [c]),
               SEEK:([c].[C]=[A].[H].[E].[C] as [e].[C]) ORDERED FORWARD)
```

For our framework, we choose a different approach to traditional query plans and introduce variable orders, which dictate the order in which we solve each join variable. Our choice is motivated by the complexity of join evaluation: standard (relation-at-a-time) query plans are provably suboptimal, whereas variable orders can be optimal [NRR13].

Definition 2.3.2 Given a join query Q, a variable X depends on a variable Y if both are in the schema of at least one relation in Q.

Definition 2.3.3 (taken from [NO18]) A variable order ω for a join query Q is a pair (F, dep), where F is a rooted forest with one node per variable in Q, and dep is a function mapping each variable X to a set of variables in F. It satisfies the following constraints:

- For each relation in Q, its variables lie along the same root-to-leaf path in F .
- For each variable X, $dep(X)$ is the subset of its ancestors in F on which the variables in the subtree rooted at X depend.

Chapter 3

Factorized Incremental View Maintenance (F-IVM)

Summary

In this chapter we introduce the Incremental View Maintenance (IVM) problem and briefly explain how F-IVM attempts to solve it. The algorithms for constructing view trees and delta view trees presented later in this chapter are situated at the very core of the computation phase of our system.

3.1 Running example

We will describe F-IVM and each module (component) of our system using the following example:

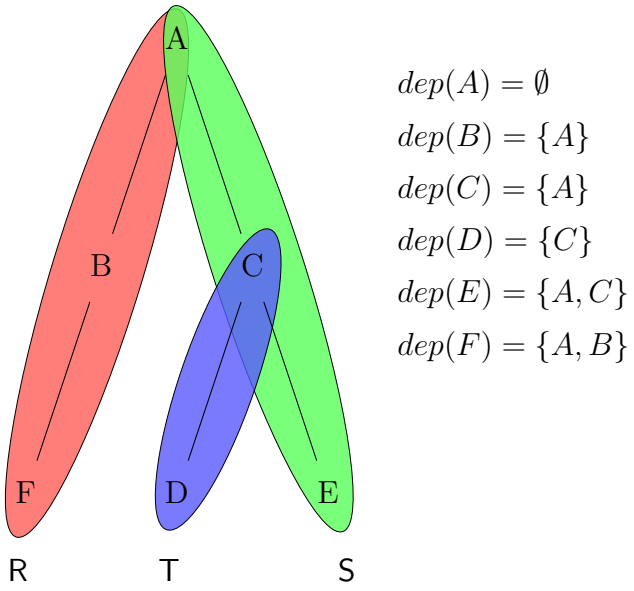


Figure 3.1: Variable order ω of the natural join of R, S, T.

We define schema $\mathcal{S}$ to be the set that contains the 6 variables, A, B, C, D, E and F. Our database $\mathcal{D}$ consists of three relations, $R(A, B, F)$, $T(C, D)$ and $S(A, C, E)$ that map tuples over $\mathcal{S}$ to elements from some fixed ring $(\mathcal{I}, +, \times, \mathbf{0}, \mathbf{1})$.
Figure 3.1 contains a variable order (also called **dtree** because of its tree-shaped representation) for the natural join of these three relations. Variable D has two ancestors, A and C, yet it only depends on one of them, C, since C and D appear together in the same relation T. There is no relation that contains both C and A so C does not depend on A (or A on C). In this case, we say that D and E are independent of each other with respect to C. The same cannot be said about C and E or B and F, for example.
On the right of the variable order, we list the dependencies of each variable. We will later use these dependencies to construct a (delta) view tree for this variable order.

3.2 Incremental View Maintenance (IVM)

A large category of datasets evolves through changes that are quite small relative to the overall dataset size. Recomputing from scratch a query over such datasets after every small change is incredibly wasteful and inefficient.

Definition 3.2.1 In a database, a **materialized view** is a database object that contains the results of a query. This view behaves like a regular table.

3.2.1 IVM

Incremental View Maintenance combines the results of previous runs (of the same query) with incoming changes to provide a computationally cheaper method for updating the result. The underlying assumption that these changes are relatively small allows us to avoid re-evaluation of expensive operations, like joins. The two extreme approaches to IVM are the first-order IVM which does not use extra views and computes changes on the fly [CY12] and the fully-recursive IVM (the approach used by DBToaster [KAK$^+$14]) which ends up computing and maintaining way too many views. F-IVM resides somewhere in the middle of this spectrum making use of a much more conservative (and efficient) approach.

3.2.2 Factorized Incremental View Maintenance (F-IVM)

This section introduces F-IVM as well as two algorithms for computing view trees and delta view trees. Both algorithms are implemented in the $ViewTree$ component of our

system. This section follows approximately the same notations, structure and content as introduced in [NO18].

F-IVM is a is a higher-order IVM algorithm that reduces the problem of maintaining an arbitrary query with joins, projections, and group-by aggregates to maintaining a hierarchy of increasingly simpler views. In our case, views are functions that map tuples over the database schema to elements from a task-specific ring. Whereas the computation over the keys is the same for all tasks, the computation over the payloads depends on the task. This algorithm may obtain a significantly lower time (and memory) complexity by factorizing the computation of the updates, keys and payloads.

Example 3.2.2 Let us consider the following SQL query Q over tables $R(A, B, F)$, $T(C, D)$ and $S(A, C, E)$:

```
SELECT SUM(R.F * T.C)
FROM R NATURAL JOIN T NATURAL JOIN S;
```

Let us consider two different evaluation strategies for this query. A naive approach first computes the join result and then the aggregate. Assuming all relations contain $O(n)$ tuples, this strategy will take $O(n^3)$ time. This is way too inefficient for most datasets if we also want to support updates!

An alternative evaluation strategy takes advantage of the distributivity of $+$ (used by the SUM operator) over multiplication and partially push the aggregate past joins. Later, this strategy will need the somehow combine this partial aggregates in order to produce the result for the given query. In the next two sections we describe how partial aggregate (called views in our framework) are computed, combined and used in order to efficiently maintain the result of Q when we perform an update to one of the relations.

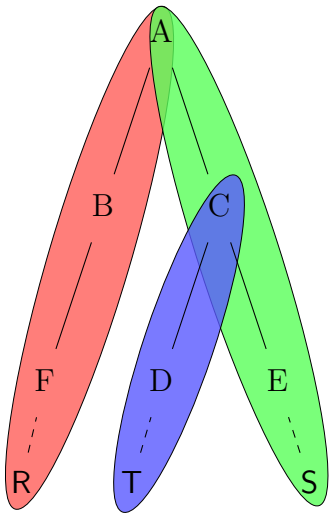


Figure 3.2: Extended variable order ω of the natural join of R, S, T.

3.2.3 Algorithm for computing view trees

Our system relies on a variable order ω to describe the structure of computation for the input query Q and to indicate which variable marginalizations are pushed past joins. At each variable $v \in \omega$ we define a view, which is simply a query over v's children. The view at the root variable corresponds to the entire query Q. We call this tree of views a view tree and we use it as the query plan in our evaluation.

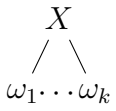
We extend the input variable order ω to include the relations, at leaves under their lowest variable. Therefore, the views in the view tree are defined over the input relations or the views of children. Figure 3.2 gives the variable order obtained after extending the one from Figure 3.1.

Figure 3.3 gives an algorithm for constructing a view tree τ given the extended variable order ω for the input query Q and the set $\mathcal{F}$ of free variables of Q.

τ (variable order ω, free variables $\mathcal{F}$)

switch ω:

R	R[schema(R)]



$$\mathsf{V}^{@\mathsf{X}}_{\mathsf{rels}}[keys]$$

$$\tau(\omega_1, \mathcal{F}) \quad \cdots \quad \tau(\omega_k, \mathcal{F})$$

, where

let

$\mathsf{V}^{@\omega_i}_{\mathsf{rels_i}}[keys_i] =$ root of $\tau(\omega_i, \mathcal{F}), 1 \le i \le k$

$keys = dep(X) \cup (\mathcal{F} \cap \bigcup_{1 \le i \le k} keys_i)$,

$\mathsf{rels} = \bigcup_{1 \le i \le k} \mathsf{rels}_i$,

$V[keys] = \bigotimes_{1 \le i \le k} \mathsf{V}^{@\omega_i}_{\mathsf{rels_i}}[keys_i]$,

in

$$\mathsf{V}^{@\mathsf{X}}_{\mathsf{rels}}[keys] = \begin{cases} V[keys] & , \text{ if } X \in \mathcal{F} \\ \bigoplus_X V[keys], & \text{otherwise.} \end{cases}$$

Figure 3.3: Algorithm that creates a view tree $\tau(\omega, \mathcal{F})$ for a variable order ω and a set of free variables $\mathcal{F}$ (taken from [NO18])

Notation: We use $\mathsf{V}^{@X}_{rels}[keys]$ to state that the view V at the variable $X \in \omega$ is recursively defined over the input relations $rels$ and has free variables $keys$. When we talk about views defined at input relations instead of variables, we use the simplified notation $\mathsf{R}[\mathsf{schema}(R)]$ to mean the view defined at R.

There are two types of nodes in the extended variable order so our algorithm will contain two cases:

- Base case (leaf node in ω): This case corresponds to an input relation R. We simply construct a view that is the relation R itself.
- Variable case (inner node in ω): This case corresponds to a variable X. We split this cases into two separate subcases:
  - If X is a bound variable, we construct a view that marginalizes out X in the natural view of its children's views. We do that by first joining on X, applying the lifting function for X on all its occurrences and then aggregating X away.
  - If X is a free variable, we retain it in the view's schema without applying the lifting function to its values or aggregating it away.

The schema of the view $V^{@X}$ consists of those ancestor variables of X in ω on which it depends and the free variables of its children (i.e. $\mathsf{schema}(V^{@X}) = keys_{V^{@X}} = dep(X) \cup (\mathcal{F} \cap \bigcup_{c \in children(X)} keys_{V^{@c}})$).

If we run our algorithm with variable order ω and $\mathcal{F} = \emptyset$ we get the following view tree:

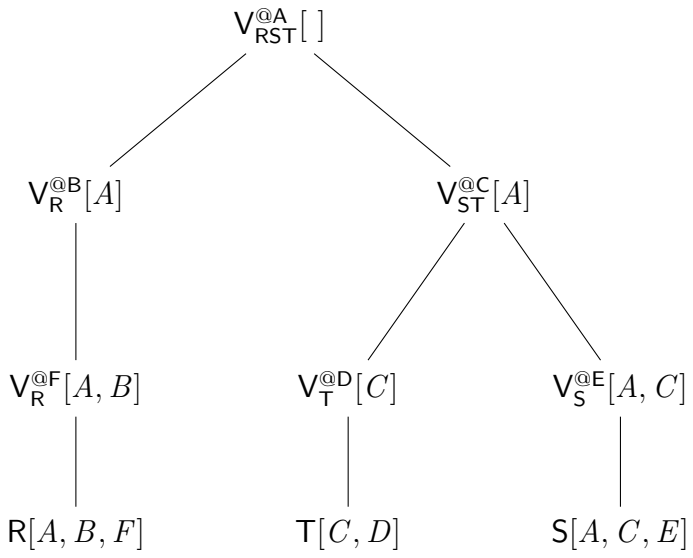


Figure 3.4: View tree τ over ω and $\mathcal{F} = \emptyset$

Using the view tree τ we can easily construct the definition of any view. For example, the view $\mathsf{V}^{@\mathsf{A}}_{\mathsf{RST}}[\,]$ (which corresponds to the result of Q) is defined as the product of its two children, $\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A]$ and $\mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]$, over which we aggregate away A. Therefore, we can say that $\mathsf{V}^{@\mathsf{A}}_{\mathsf{RST}}[\,] = \bigoplus_A \left(\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A] \otimes \mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]\right)$. Notice that for $\mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]$ we also include C in the product. That is because variable C is present in the product used as the argument of the SUM operator in query Q. Same argument for F in $\mathsf{V}^{@\mathsf{F}}_{\mathsf{R}}[A, B]$.

Using the same mechanical approach we can get the definitions for all our views $V \in \tau$:

- $\mathsf{V}^{@\mathsf{A}}_{\mathsf{RST}}[\,] = \bigoplus_A \left(\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A] \otimes \mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]\right)$
- $\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A] = \bigoplus_F \mathsf{V}^{@\mathsf{F}}_{\mathsf{R}}[A, B]$
- $\mathsf{V}^{@\mathsf{F}}_{\mathsf{R}}[A, B] = \bigoplus_B \left(\mathsf{R}[A, B] \otimes F\right)$
- $\mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A] = \bigoplus_C \left(\mathsf{V}^{@\mathsf{D}}_{\mathsf{T}}[C] \otimes \mathsf{V}^{@\mathsf{E}}_{\mathsf{S}}[A, C] \otimes C\right)$
- $\mathsf{V}^{@\mathsf{D}}_{\mathsf{T}}[C] = \bigoplus_D \mathsf{T}[C, D]$
- $\mathsf{V}^{@\mathsf{E}}_{\mathsf{S}}[A, C] = \bigoplus_E \mathsf{S}[A, C, E]$

3.2.4 Algorithm for computing delta view trees

In this section we introduce the notions of updates, delta views and delta view trees and explain their usage in incrementally maintaining views in our framework. We end this section with an algorithm for computing delta view trees.

Updates. We express the insertion (deletion) of a tuple $\mathbf{t}$ into (from) a relation R as a delta relation δR that maps t to $\mathbf{1}$ ($-\mathbf{1}$). We can then decompose the updated relation into the union of the old relation and the delta relation: $\mathsf{R} := \mathsf{R} \uplus \delta\mathsf{R}$.

Delta Views. For each view V affected by an update, a *delta view* $\delta\mathsf{V}$ defines the change in the view contents. We define $\delta\mathsf{V}$ by case analysis on V:

- If V represents a relation R, then $\delta\mathsf{V} = \delta\mathsf{R}$ if $R \in \mathcal{U}$ (i.e. R is updatable) and $\delta\mathsf{V} = \emptyset$ otherwise.
- If V does not represent a relation, then V is defined in terms of other views. We can derive $\delta\mathsf{V}$ using the following delta rules:

$$\begin{aligned}
\delta(\mathsf{V}_1 \uplus \mathsf{V}_2) &= \delta\mathsf{V}_1 \uplus \delta\mathsf{V}_2 \\
\delta(\mathsf{V}_1 \otimes \mathsf{V}_2) &= (\delta\mathsf{V}_1 \otimes \mathsf{V}_2) \uplus (\mathsf{V}_1 \otimes \delta\mathsf{V}_2) \uplus (\delta\mathsf{V}_1 \otimes \delta\mathsf{V}_2) \\
\delta(\textstyle\bigoplus_X \mathsf{V}) &= \textstyle\bigoplus_X \delta\mathsf{V}
\end{aligned}$$

Note: If V is not defined over the updated relation R, then $\delta\mathsf{V} = \emptyset$ and we use the following identities $\emptyset \uplus \mathsf{V} = \mathsf{V} \uplus \emptyset = \mathsf{V}$ and $\emptyset \otimes \mathsf{V} = \mathsf{V} \otimes \emptyset = \emptyset$.

Delta Trees. Under updates to one relation, a view tree becomes a delta tree where the affected views become delta views. The algorithm in Figure 3.5 traverses the view tree τ top-down along the path from the root to the updated relation R and creates a delta path (i.e. a path where all views are replaced with delta views).

Figure 3.5 gives an algorithm for constructing a delta view tree $\Delta(\tau, \delta R)$ given the view tree τ and the update δR to the relation R.

Δ (view tree τ, update $\delta\mathsf{R}$)

switch τ:

$\mathsf{R}[\mathsf{schema}(\mathsf{R})]$	$\delta\mathsf{R}[\mathsf{schema}(\mathsf{R})]$

$\mathsf{V}^{@\mathsf{X}}_{\mathsf{rels}}[keys]$ — $\tau_1 \cdots \tau_k$

$\delta\mathsf{V}^{@\mathsf{X}}_{\mathsf{rels}}[keys]$ — $\tau_1 \cdots \Delta(\tau_j, \delta\mathsf{R}) \cdots \tau_k$, where

let

$\mathsf{V}^{@\tau_i}_{\mathsf{rels_i}}[keys_i]$ be root of $\tau_i, 1 \le i \le k$,
be such that $\mathsf{R} \in \mathsf{rels}_j$ and $1 \le j \le k$,
$\delta V[keys] = \delta\mathsf{V}^{@\tau_j}_{\mathsf{rels_j}}[keys_j] \otimes \bigotimes_{1 \le i \ne j \le k} \mathsf{V}^{@\tau_i}_{\mathsf{rels_i}}[keys_i]$,

in

$$\delta\mathsf{V}^{@\mathsf{X}}_{\mathsf{rels}}[keys] = \begin{cases} \delta V[keys] & , \text{ if } X \in \mathcal{F} \\ \bigoplus_X \delta V[keys] & , \text{ otherwise.} \end{cases}$$

Figure 3.5: Algorithm that creates a delta view tree $\Delta(\tau, \delta\mathsf{R})$ for a view tree τ to accommodate an update $\delta\mathsf{R}$ to relation R (taken from [NO18]).

We notice that the algorithm is very similar to the one for constructing view trees. The main difference is that we create and use delta views along the delta path for R. We can compute tree delta views (one for F, one for F and one for A) using the same approach we used in computing view trees. We get:

- $\delta\mathsf{V}^{@\mathsf{A}}_{\mathsf{RST}}[\,] = \bigoplus_A \left(\delta\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A] \otimes \mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]\right)$
- $\delta\mathsf{V}^{@\mathsf{B}}_{\mathsf{R}}[A] = \bigoplus_F \delta\mathsf{V}^{@\mathsf{F}}_{\mathsf{R}}[A, B]$
- $\delta\mathsf{V}^{@\mathsf{F}}_{\mathsf{R}}[A, B] = \bigoplus_B \left(\delta\mathsf{R}[A, B] \otimes F\right)$

Returning to our initial time complexity analysis, for an update on R it only takes linear time to recompute the answer of Q assuming we materialize (cache) view $\mathsf{V}^{@\mathsf{C}}_{\mathsf{ST}}[A]$. This is clearly a huge improvement over the naive, cubic time solution.

Chapter 4

System implementation

Summary

In this chapter we give a detailed overview of our system while presenting the transformation from the theoretical approach to IVM introduced in *Chapter 3: F-IVM* into functional, fully-tested Scala code. Figure 4.1 gives a visual representation of our pipeline.

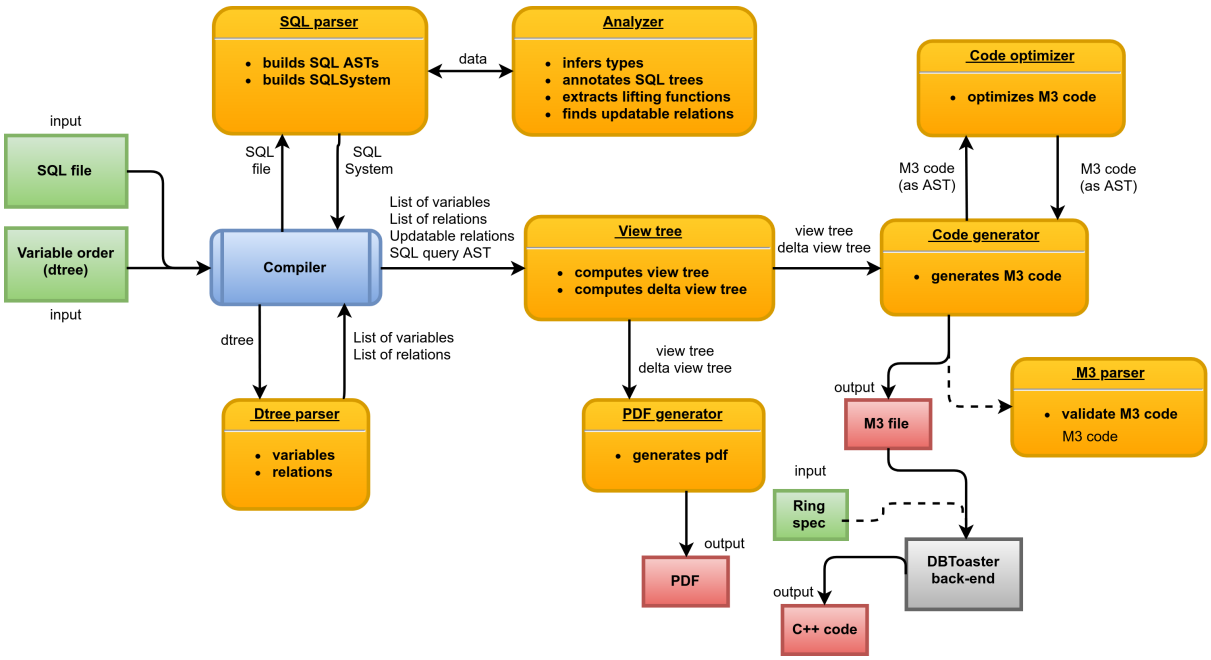


Figure 4.1: Component Diagram of our system

4.1 DBToaster

DBToaster is a generative compiler that takes a SQL query (which the user wants to incrementally maintain under a stream of updates) and generates the incremental view maintenance program (IVM) as a C++ or Scala executable file.

The freshly generated IVM program can be then plugged into the user's own application(s). It only needs the data stream, which consists of all insert, update and delete operations on the (dynamic) data tables.

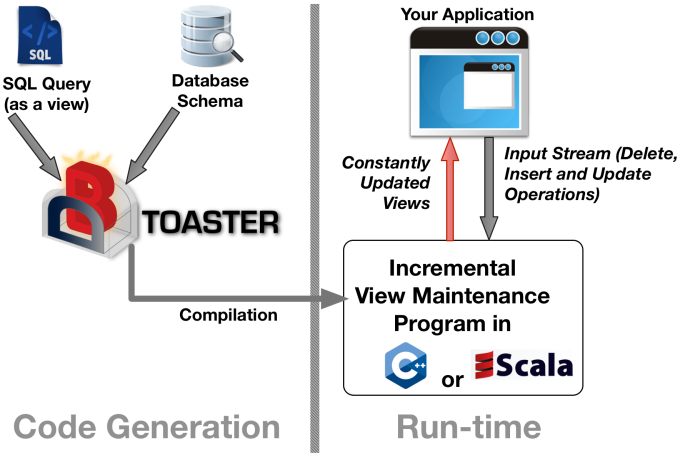


Figure 4.2: DBToaster process (taken from https://dbtoaster.github.io/)

The DBToaster is internally divided into two main components: front-end and back-end.

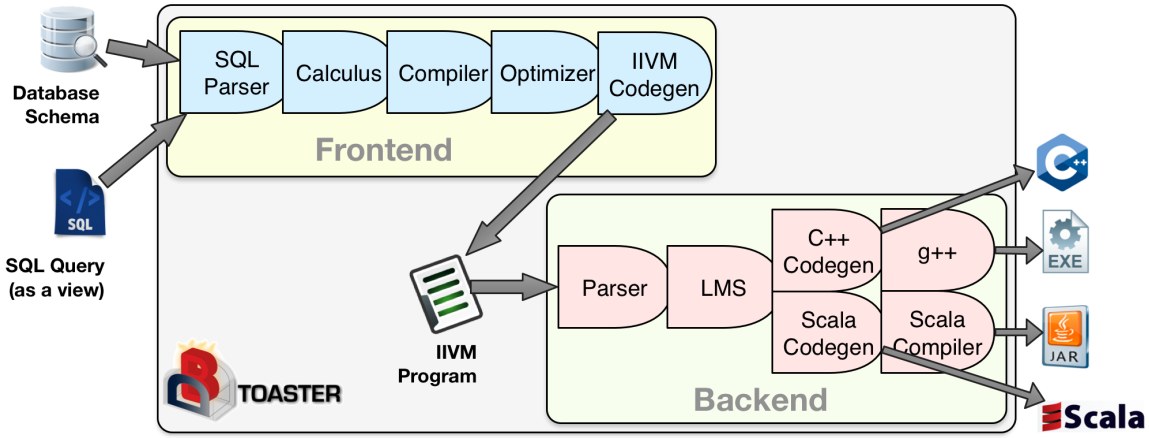


Figure 4.3: DBToaster architecture (taken from https://dbtoaster.github.io/)

As it is presented in Figure 4.3, the task of front-end is to parse the given SQL program and convert it into an internal calculus for IVM and then it is compiled and optimized and finally converted into M3, an intermediate IVM language that is specific to DBToaster.

The back-end will then take the M3 code generated by the front-end and after parsing and optimizing it using Lightweight Modular Staging (LMS), a framework for runtime code generation and compiled DSLs, it will then use either the C++ or the Scala code generator to produce the concrete output program in C++ or Scala, respectively.

Our system will replace the entire front-end of DBToaster, generate highly-efficient, optimized, back-end compatible M3 code using the Factorized Incremental View Maintenance framework and feed the code into a slightly modified version of DBToaster's back-end.

4.2 Input files

A typical run of our system will require two input files (three if we want to use a custom ring).

The first input file is the SQL file which contains all the necessary information about the tables/streams (both their definition and where we can load them from), whether or not there is a ring file (and a custom type created for it) and the SQL query.
Below you can see a rather typical SQL input file (which we will add to our running example):

Listing 4.1: UPDATE_R.sql

```
CREATE TYPE Ring
FROM FILE "'ring.hpp'";

CREATE STREAM R(A int, B double, F double)
FROM FILE './datasets/thesis/R.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE S(A int, C int, E int)
FROM FILE './datasets/thesis/S.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE T(C int, D int)
FROM FILE './datasets/thesis/T.tbl' LINE DELIMITED CSV(delimiter := '|');

SELECT SUM([f_A: Ring](A) * [f_B: Ring](B) * [f_F: Ring](F) * [f_D: Ring](D))
FROM R NATURAL JOIN S NATURAL JOIN T
```

One can easily notice that the SQL file is of the correct format: it contains the tables (static relations), S and T, the stream (dynamic relation), R, the query which uses four external functions (the lifting functions for A, B, F and D), the location of a C++ header file that contains the specification of our ring (including the actual implementations of $f_A : Domain(A) \to \mathcal{I}$, $f_B : Domain(B) \to \mathcal{I}$, $f_F : Domain(F) \to \mathcal{I}$

and $f_D : Domain(D) \rightarrow \mathcal{I}$) and a custom type '*Ring*' that is going to be integrated into our type annotater/checker.

The second non-optional input file is the variable order (i.e. the dtree). This file contains the number of variables and relations followed by the list of all variables along with their immediate parent, dependencies and types and a boolean field. The file ends with the description of all relations (both static and dynamic) present in the SQL file.

Listing 4.2: dtree.txt

```
6 3
0 A int -1 {} 0
1 B double 0 {0} 0
2 C int 0 {0} 0
3 D int 2 {2} 0
4 E int 2 {0,2} 0
5 F double 1 {0,1} 0
R 5 A,B,F
S 4 A,C,E
T 3 C,D
```

The (optional) third input file is the ring specification (as a C++ header file). An example of a ring can be seen in *Chapter 5: Applications*.

4.3 SQL parser

In this section we describe the SQL parser, a module that builds an abstract syntax tree (AST) from a SQL query and wraps the whole SQL file into a custom unit called SQLSystem (that is later used to pass relevant information to the next stage of our pipeline).

Our SQL parser is a modified version of DBToaster's parser which can be found in DBToaster's Github repository (https://github.com/dbtoaster/dbtoaster-a5). Both the SQL and the M3 parser (defined in a later section) are $StandardTokenParsers$ from a packaged called *scala-parser-combinators.jar*, which is the only dependency of our system.
The latest stable version of DBToaster (2.2 as of May 2018) does not support custom types (i.e. rings) so we had to extend the parser in order to accommodate the new syntax. We use case classes to represent nodes as we would like to use the technique of pattern matching to differentiate between separate types of expressions. By using pattern matching, we can easily deconstruct our trees and extract relevant information

when needed in a clean and readable way. We have made the following changes to the SQL parser:

Listing 4.3: SQLParser.scala

```
lazy val funct =
  ("[" ~> /* func */ ident <~ ":") ~ (tpe <~ "]") ~
  ("(" ~> repsep(field,",") <~ ")") ^^ {
  case n ~ t ~ as => FunApply(n, TypeFunction(t), as)
}

val typeMap = collection.mutable.HashMap.empty[String, TypeDefinition]

lazy val typeDef =
 "CREATE" ~> "TYPE" ~> ident ~ ("FROM" ~> "FILE" ~> stringLit) <~ ";" ^^ {
  case i ~ f =>
    val t = TypeDefinition(i, f)
    typeMap += ((i, t))
    t
}

lazy val customType: Parser[TypeCustom] =
 acceptIf (x => typeMap.contains(x.chars) ) (x => "No such type '" + x.chars
     + "'") ^^ {
  case i => TypeCustom(i.chars, typeMap(i.chars))
}
```

We also extended the abstract syntax tree with nodes that represent function application ($FunApply$), type definitions and custom types. By source code inspection, one can notice that if $e = [f : some\_type](X_1, X_2, ..., X_n)$ is a syntactically valid SQL (sub)expression then e will be parsed into a

$$FunApply(f, \text{parse some\_type}, \text{map parse}[X_1, X_2, ..., X_n])$$

node in the abstract syntax tree. Type definitions and custom types also have their own nodes (as case classes) as part of the SQL AST hierarchy.

Listing 4.4: SQLTree.scala

```
// Function application (possibly with custom types)
case class FunApply(fun: String, _tp: Type, args: List[Expr]) extends Expr {
  override def tp: Type = _tp
  override def toString: String = "[" + fun + ": " + tp + "](" +
      args.mkString(", ") + ")"
}

case class TypeDefinition(val name: String, val path: String) {
  ...
}
case class TypeCustom(name: String, typeDef: TypeDefinition) extends Type {
  ...
}
```

4.4 Abstract syntax trees analyzer

Any respectable code-generation framework that uses a type system must contain a module responsible for deducing (inferring) composite types from ground ones and annotating AST nodes with relevant information.
Therefore, in this section we describe the AST Analyzer, a module which contains a small type-inference system and is in charge of analyzing SQL ASTs, annotating them with types and extracting the definitions of all lifting functions present in such a tree.

The type inference system is implemented using two methods

$$resolve(b : Type) : Type$$

which tries to solve a type conflict if possible using type promotion or throws a type-mismatch exception and

$$combineTypes(l : Type, r : Type) : Type$$

which is used internally by AST nodes to get the resulting type from combining the types of their children.

The method *resolve* is present in all nodes which represent primitive types. In Listing 4.5 we show the implementation of *resolve* for $TypeShort$. If b is of type *char* or *short* then we 'promote' (if necessary) b and return $TypeShort$. If b is of type *int*, *long*, $float$ or *double* then we 'promote' 'this' and return b's type. If neither case was chosen, then we report a $TypeMismatchException$ since we know that there are no other type conversions allowed for $TypeShort$.

Listing 4.5: TypeShort

```
case object TypeShort extends Type {
  def resolve(b: Type) = b match {
    case TypeChar | TypeShort => this
    case TypeInt | TypeLong | TypeFloat | TypeDouble => b
    case _ => throw TypeMismatchException("Type mismatch (" + this + ", " + b
        + ")")
  }
  override def toString = "short"
}
```

In Listing 4.6 you can see a short fragment from the AST file that shows how *SQL.Add* derives its type from its two operands.

Listing 4.6: SQLTree.scala

```
case class Add(l: Expr, r: Expr) extends Expr {
    override def tp: Type = Analyzer.combineTypes(l.tp, r.tp)
}
```

The method *combineTypes* (Listing 4.7) is quite easy to implement once we decide the order (priority) of all type deduction rules. We can observe that is was a great design choice to use case classes to represent nodes as we can now easily deconstruct and pattern match on our types.
There are two basic (symmetric) cases that involve *null* values. Another two for *TypeFunction* (which is the type of external functions), one for combining two custom types (if there are the same) and one for combining something with a custom type.
A last case which defaults to *resolve* is added in order to allow usage of primitive types (e.g. *int*, *date*, *short*).

Listing 4.7: Analyzer.scala

```
def combineTypes(l: Type, r: Type): Type = (l, r) match {
  case (`l`, null) => l
  case (null, `r`) => r
  case (TypeFunction(tp), other) => combineTypes(tp, other)
  case (other, TypeFunction(tp)) => combineTypes(other, tp)

  case (TypeCustom(name1, _), TypeCustom(name2, _)) =>
    if (name1.equals(name2))
      l
    else
      throw new TypeMismatchException

  case (`l`, customType @ TypeCustom(_, _)) => combineTypes(customType, l)
  case (`l`, `r`) => l.resolve(r)
}
```

We also need extra functionality in order to extract all the occurrences of all lifting functions present in the SELECT clause of a given SQL query since lifting functions are external (i.e. their actual implementations are in a completely different file). In order to do just that, we have another method

$$analyzeSQLTree(expr : SQL.Expr, funMap : Map[SQL.Field, SQL.Expr], \\ externalFunList : List[(Set[SQL.Field], SQL.FunApply)])$$

Note that $analyzeSQLTree$ takes two function-related arguments, $funMap$ which maps a variable (SQL.Field node) to its explicit function (see the example below), and $externalFunList$ which contains a list with all occurrences of lifting functions. Also, note that our implementation supports multivariate lifting functions that map a set of variables (set of keys) to a payload.

The last thing we do is check for the existence of a GROUP BY clause. If such a clause exists, we extract the set of free variables and pass them to the next stage.

Example 4.4.1 Let us consider the following SQL query:

Listing 4.8: example.sql

```
CREATE TYPE Ring
FROM FILE "'ring.hpp'";

CREATE TABLE R(A int, B double)
FROM FILE './datasets/thesis/R.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE S(A int, C int, E int)
FROM FILE './datasets/thesis/S.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE T(C int, D int)
FROM FILE './datasets/thesis/T.tbl' LINE DELIMITED CSV(delimiter := '|');

SELECT E, D, SUM([f_A: Ring](A) * (A * 2) * (B + 1) * C * C)
FROM R NATURAL JOIN S NATURAL JOIN T
GROUP BY E, D
```

Running our analyzer on the query above we get the following .log file:

Listing 4.9: analyzed.log

```
----------------------ANALYZE QUERY--------------------------
SELECT E, D, SUM(((((f_A: Ring](A) * (A * 2)) * (B + 1)) * C) * C))
FROM R
  NATURAL JOIN S
  NATURAL JOIN T
GROUP BY E, D

FunMap: Map(A -> (A * 2), C -> (C * C), B -> (B + 1))
ExternalFunList: List((Set(A),[f_A: Ring](A)))
Free Variables: Set(E, D)
```

After inspecting the .log file one can notice several curious facts:

- our parser appears to be a LL parser which produces a leftmost derivation of the given SQL expression
- there are two types of functions:
  - explicit: $A \to A * 2$ or $C \to C^2$
  - implicit (or external) which is the lifting function f_A
- the implementation of lifting functions is indeed multivariate, f_A takes a singleton set instead of a plain variable
- the $+$ and $*$ in $B + 1$ or $C * C$ are the arithmetic ones but the $*$ between $(A * 2)$ and $(C * C)$ is the $\times$ from the ring

4.5 DTree parser

This tiny parser takes a variable order input file and produces one list of variables and one of relations. Both lists are then passed to the next component, which uses them to build the (delta) view tree(s).
There are two concrete classes in a package unsurprisingly named *dtree* which are used to implement a variable and a relation respectively.

Listing 4.10: package **dtree**

```
class Variable(val id: Int, val name: String, val vtype: Type,
               val parent: Int, val deps: List[Variable], val toCache: Boolean)

class Relation(val name: String, val variables: List[Variable],
               var lowestVariable: Variable)
```

For each variable, we store its id (the order in dtree.txt), name, type, parent's id, list of dependencies and whether or not its view should be cached (i.e. materialized) in the view tree. The field *toCache* is used as a hint for our compiler when deciding which views must be materialized in order to support efficient updates. Note that *toCache* represents only a hint. The compiler can choose to ignore it if this would lead to a more time or memory efficient solution. The names and types must be consistent with the ones used in the SQL file and a special check is performed in the *Compiler* component to enforce this constraint.

For each relation, we store its name, variables and the lowest variable (by depth) in the dtree (if we were to treat it as a regular rooted tree). The lowest variable field is only relevant to this specific implementation as we construct all delta paths using a bottom-up approach. We chose to include this piece of information in the variable order file in order to simplify our implementation. Another design option would have been to do a depth-first-search traversal of the dtree and store for each relation the lowest variable encountered so far in the traversal.

4.6 Compiler

This module represents the central component of our framework. Unlike all other modules described in this chapter, this one does not perform any real computation on the given data. The SQL input file is passed to the SQL parser while the variable order input file is sent to the DTree parser. Both results are then collected, wrapped nicely in a custom-made unit and fed into the $ViewTree$ module, where all the magic happens.

This component is also in charge of parsing the flags and ensuring that all other components obey them (for example, if optimizations are turned off then the code generator must not use the code optimizer in any way; another component which makes good use of flags is the PDF generator).

A special output stream is created and connected to a .log file in order to record different events or messages produced at every stage of our pipeline thus providing an audit trail that can be used to understand the activity of the system or to diagnose problems. Logging is essential to understand the activities of such a complex system, particularly since it involves minimal user interaction. Each run (even an unsuccessful one) of our system produces a unique log file.

4.7 ViewTree

This module is responsible for computing view trees and delta view trees using the two algorithms introduced in *Chapter 3: Factorized Incremental View Maintenance.*

4.8 Code generation

Once we have computed both the view tree and the delta view trees for our set of updatable relations $\mathcal{U}$, we need a way of converting our definitions of views into actual valid M3 code. This module achieves that by creating M3 triggers for all relations in $\mathcal{U}$.

Definition 4.8.1 A trigger is procedural code that is automatically executed in response to certain events on a particular table or view in the database.

DBToaster supports a wide variety of triggers but we will only focus on two: 'batch updates' and 'on system'. The 'on system' trigger is unique and called only once when the application is run. It is generally used to initialize static relations and to update all maps that only depend on those relations. The other kind is used when we perform multiple updates (insertion/deletions) on some relation. Note that we group our updates into batches of different sizes (thus placing no restriction on the order of records in input relations) in order to improve the IO performance.

IVM batch triggers. For each updatable relation $R \in \mathcal{U}$, our framework constructs a trigger which takes as input an update δR, updates R while efficiently maintaining the corresponding delta view tree.

Figure 4.4 gives a procedure for computing an IVM batch triggers for an updatable relation R. We start from R's lowest variable and continue on the delta path until we hit the root. At each stage, we check if the current variable needs caching. If it does then we create two M3 statements that update the two maps using the value of $\delta V^{@X}$. Note that the algorithm requires knowledge of all maps in our program and also that we know precisely which views must be materialized and maintained.

Input: δR
Output: IVM batch trigger for R

```
1  let X = R.lowest_variable;
2  let triggers = List.empty;
3  while X ≠ null do
4  |  compute δd as the definition of δV^@X;
5  |  if X.toCache = true then
6  |  |  /* initialize the corresponding delta map with δd                */
7  |  |  triggers += M3.Stmt(DELTA_AGG_VIEW_X[] := δd));
8  |  |  /* update X's normal map by incrementing it with δd (which is
   |  |     now stored in the delta map)                                   */
9  |  |  triggers += M3.Stmt(AGG_VIEW_X[] += DELTA_AGG_VIEW_X[]));
10 |  end
11 |  X := X.parent;
12 end
13 return triggers
```

Figure 4.4: Procedure for generating batch triggers

Example 4.8.2 By running the algorithm in Figure 4.4 on relation R from our running example we get the following batch trigger:

Listing 4.11: Batch update trigger for R

```
ON BATCH UPDATE OF R {
  DELTA_AGG_VIEW_B(Ring)[][A] := AggSum([A],
    ((DELTA R)(A, B, F) * ([f_B: Ring](B) * [f_F: Ring](F)))
  );
  AGG_VIEW_B(Ring)[][A] += DELTA_AGG_VIEW_B(Ring)[][A];
  DELTA_AGG_VIEW_A(Ring)[][] := AggSum([],
    ((DELTA_AGG_VIEW_B(Ring)[][A] * AGG_VIEW_C(Ring)[][A]) * [f_A: Ring](A))
  );
  AGG_VIEW_A(Ring)[][] += DELTA_AGG_VIEW_A(Ring)[][];
}
```

How to decide which views to materialize? It is clearly obvious that we must materialize all views that refer to a static relation. Otherwise, we would have to load multiple tuples from memory each time we perform some operation involving that relation. But what about all other views? The answer to this question depends on which relations may change. The set of updatable relations uniquely determines the set of delta propagation paths in a view tree, and these paths might require some views to be materialized. General idea: According to [NO18], propagating changes along a leaf-to-root path is computationally most effective if each delta view joins with sibling views that are already materialized.

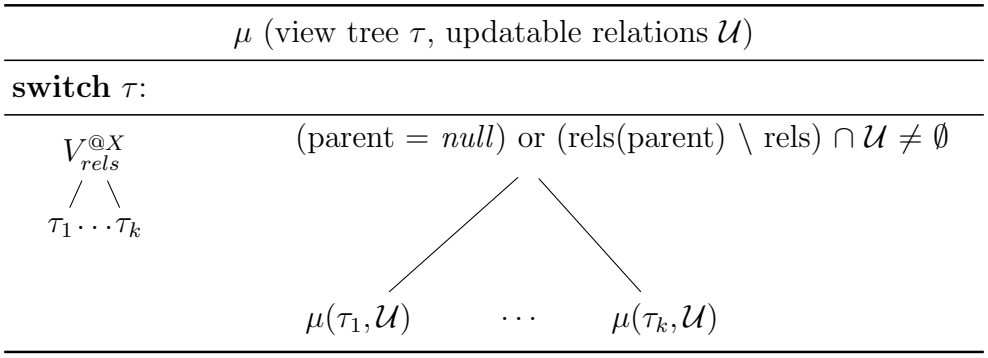


Figure 4.5: Algorithm that decides which views in a view tree τ to materialize in order to support updates to a set of relations $\mathcal{U}$. The notation $\mathsf{rels}(x)$ denotes the relations over which x is defined.

Figure 4.5 gives an algorithm that uses the idea introduced in the previous paragraph. Given a view tree τ and a set of relations $\mathcal{U}$, the algorithm computes for each view a boolean value which is true if the view must be materialized and false otherwise. In the actual implementation, we use the field *toCache* to store the value.
The root view is always materialized as it represents the query result (the root has no parent so it hits the first case). Every other view V is stored only if it is used to compute the delta of its parent for updates to a relation over which V is not defined (simply put, there are updatable relations for the parent and not for V itself: (rels(parent) \ rels) ∩ $\mathcal{U} \neq \emptyset$).

After we have applied the algorithm from Figure 4.5 we need to traverse the view tree one more time, create and collect all the maps. For:

- $R[schema(R)]$ we create the map $VIEW\_R(long)[schema(R)]$
- $V^{@X}_{rels}[keys]$ we create the map $AGG\_VIEW\_X(view\_type)[typed\_keys]$
- $\delta V^{@X}_{rels}[keys]$ we create the map $DELTA\_AGG\_VIEW\_X(view\_type)[typed\_keys]$

Example 4.8.3 Going back to our running example, we show the M3 code for declaring some of the maps created by our algorithms:

Listing 4.12: M3 declarations of maps

```
-- create a map for the static relation T
DECLARE MAP VIEW_T(long)[][C: int, D: int] := 0L;
-- create a map for V_T^@D[C]
DECLARE MAP AGG_VIEW_D(Ring)[][C: int] :=
    AggSum([C], (T(C, D) * [f_D: Ring](D)));
-- create a map for δV_T^@D[C]
DECLARE MAP DELTA_AGG_VIEW_D(Ring)[][C: int] :=
    AggSum([C], ((DELTA T)(C, D) * [f_D: Ring](D)));
```

4.9 Optimizer

Because we use a very mechanical approach to generate the IVM triggers, the resulting code may contain some inefficiencies. In this section, we introduce a module, similar to an optimizing compiler, designed specifically to eliminate some classes of inefficiencies that arise frequently in practice.

4.9.1 Long chains of views

By default, the algorithm given in Figure 3.3 constructs one view per variable v for $v \in \omega$. A wide relation (i.e. one with many variables) leads to long chains of views for variables that are only local to this relation. We compose such long chains of views into a single view that marginalizes several variables at a time. Let us consider the following SQL file:

Listing 4.13: housing.sql

```
CREATE TABLE HOUSE(postcode float, h2 float, h3 float, h4 float, h5 float, h6
    float)
FROM FILE './House.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE SHOP(postcode float, s2 float, s3 float, s4 float, s5 float, s6
    float)
FROM FILE '.Shop.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE INSTITUTION(postcode float, i2 float, i3 float)
FROM FILE '.Institution.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE RESTAURANT(postcode float, r2 float, r3 float)
FROM FILE '.Restaurant.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE DEMOGRAPHICS(postcode float, d2 float, d3 float, d4 float, d5
    float)
FROM FILE '.Demographics.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE TRANSPORT(postcode float, t2 float, t3 float, t4 float)
FROM FILE '.Transport.tbl' LINE DELIMITED CSV(delimiter := '|');

SELECT SUM(postcode * postcode * (h2 + 1) * (h3 * 5))
FROM HOUSE NATURAL JOIN SHOP NATURAL JOIN INSTITUTION NATURAL JOIN RESTAURANT
    NATURAL JOIN DEMOGRAPHICS NATURAL JOIN TRANSPORT;
```

We have six relations than only share one variable but each one of them contains other variables (that are not really relevant outside of their relation). If we are not using any optimizations, we would get a view tree similar to:
If we were to use the optimization, we would get a much more compressed view tree:

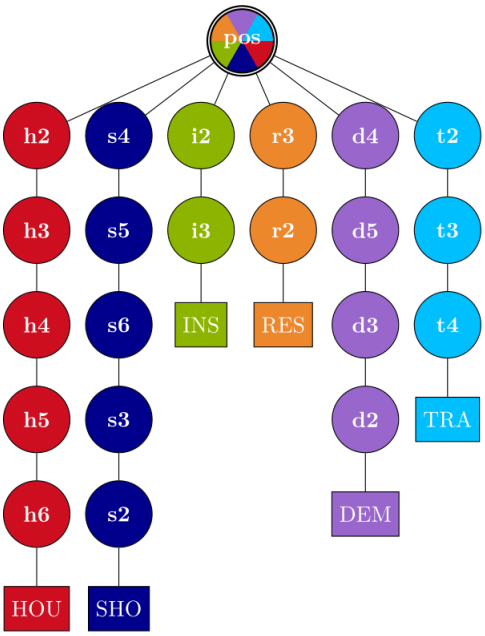


Figure 4.6: View tree with long chains

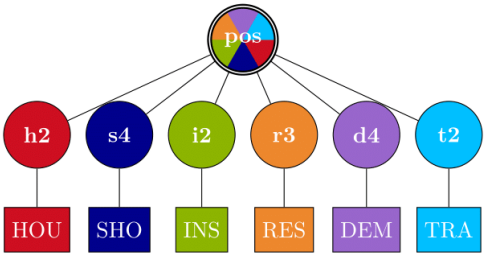


Figure 4.7: View tree with compressed chains

This optimization is not hard to implement (we can optimize away a node if and only if its parent contains only one child) but we have to pay extra attention to the functions associated with each variable. Let us consider the case that involves the variables $h2$ and $h3$. Since $h2$ only contains one child, $h3$, we can merge them together, thus eliminating $h3$ from the view tree. But that should happen with $h3$'s explicit function, $f_{h3} = (h3 * 5)$? We need to somehow update $h2$ in order not to lose this function. The solution is quite simple, set $f_{h2} := f_{h2} \times f_{h3} = (h2 + 1) \times (h3 * 5)$. This solution only works because we construct and update the delta paths from the lowest variable to the root. Thus $h3$ is updated (and its internal function used) right before $h2$ each time in both versions. The same changes have to be made for lifting functions (if any exists).

Example 4.9.1 If we use this optimization technique to our running example, we get the following compressed variable order:

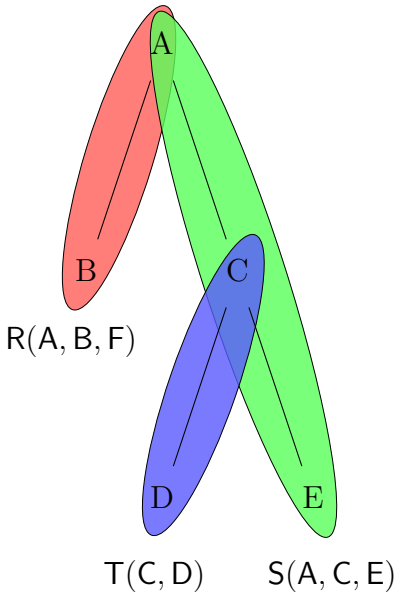


Figure 4.8: Variable order ω' obtained from compressing ω

Note that variable F is not removed from relation R or schema $\mathcal{S}$. It is only eliminated from our query plan and its explicit/implicit functions are passed to B.

4.9.2 Inline maps

There are cases when we compute something, store it in a map, and then only use the map once in some other computation. Since our maps are not trivial data structures (they are usually highly-optimized hash-maps) storing and retrieving data comes with a price. It would much more efficient (both time and memory wise) to inline the first computation into the second one. Compare the following two fragments of M3 code:

Listing 4.14: Unoptimized M3 code

```
ON BATCH UPDATE OF R {
  DELTA_AGG_VIEW_B(Ring)[][A] := AggSum([A],
    ((DELTA R)(A, B, F) * ([f_b: Ring](B) * [f_f: Ring](F)))
  );
  AGG_VIEW_B(Ring)[][A] += DELTA_AGG_VIEW_B(Ring)[][A];
  DELTA_AGG_VIEW_A(Ring)[][] := AggSum([],
    ((DELTA_AGG_VIEW_B(Ring)[][A] * AGG_VIEW_C(Ring)[][A]) * [f_a: Ring](A))
  );
  AGG_VIEW_A(Ring)[][] += DELTA_AGG_VIEW_A(Ring)[][];
}
```

Listing 4.15: Optimized M3 code

```
ON BATCH UPDATE OF R {
  DELTA_AGG_VIEW_B(Ring)[][A] := AggSum([A],
    ((DELTA R)(A, B, F) * ([f_b: Ring](B) * [f_f: Ring](F)))
  );
  AGG_VIEW_B(Ring)[][A] += DELTA_AGG_VIEW_B(Ring)[][A];
  AGG_VIEW_A(Ring)[][] += AggSum([],
    ((DELTA_AGG_VIEW_B(Ring)[][A] * AGG_VIEW_C(Ring)[][A]) * [f_a: Ring](A))
  );
}
```

We have eliminated $DELTA\_AGG\_VIEW\_A$ (and inlined its definition) since we only use it once on the right-hand side of an assignment.

In order to implement this optimization we need a way to count the number of occurrences of a map on the RHS of an assignment in a block of code. This can become close to impossible if the wrong data-encoding is used. Fortunately, our tree-based representation comes to our rescue. Our code is represented as an M3 abstract syntax tree so we can traverse it using a depth-first-search approach, deconstruct it at each stage by using pattern matching and update a counter for each map when we encounter a node that represents that type of map. Once we know which maps we have to inline, we traverse the tree a second time, generating a new one: if we encounter a map which must be inlined we simply return its definition, otherwise we return the map. All other cases (i.e. the ones not involving maps) remain the same.

Finally, the optimized code is then passed back to the code generator and the pipeline continues its work.

4.10 PDF generation

So far we have presented a lot of toy examples, using small-scale variable orders which are quite artificial in their nature. Real-life applications usually require complex schemas with dozens of relations and hundreds of variables. Being able to analyze a variable order without any tools, just be looking at it, becomes very important when deciding is some variable order ω has some given property (e.g. all free variables are on top of the bound variables) or if a view was computed correctly.

This module is used in order to provide a graphical representation of a view tree. Actually, this component creates a hybrid tree, a view tree with delta views integrated into it. After a successful run of this module, we get a LaTeX document which we can then compile (using *luatex* twice) into a PDF. The benefit of this approach is that we

only need to worry about generating .tex files and that we can use awesome packages like *tikz*, *trees* or *hyperref* in order to provide some functionally inside the PDF document.

All generated PDFs contain four sections:

- Hybrid tree diagram
- Legend with useful information
- Definitions for all views and delta views
- Abbreviation list

4.10.1 Hybrid tree diagram

Let us consider the following variable order ω:

```
7 4
0 A int -1 {} 0
1 B int 0 {0} 0
2 C int 1 {0,1} 0
3 D int 1 {0,1} 0
4 E int 0 {0} 0
5 F int 4 {4} 0
6 extra int 2 {0,1,2} 0
R 6 A,B,C,extra
S 3 A,B,D
T 4 A,E
U 5 E,F
```

We have four relations, R, S, T and U, 7 variables and we only want to support updates for R (i.e. R is a stream while S, T and U are tables). We also list the dependencies for each variable and the fact that there is no hint to materialize any of them. It is all up to our system to decide which maps to create. After running our PDF component on the view tree generated for ω we get Figure 4.9:

We can immediately notice a few things:

- The variable *extra* is not present in our hybrid tree as the variable was eliminated from the view tree during the optimization phase.
- Each relation has a unique colour. A variable that belongs to more than one relation is coloured with more than one colour. This is extremely helpful as it allows us to analyze relations effortlessly.

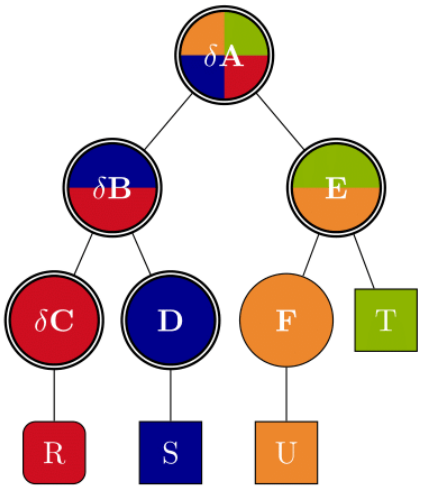


Figure 4.9: Hybrid tree

- Some nodes are circles (single and double) and some are rectangles (with and without rounded corners).
- Some nodes contain the δ (delta) symbol in front of their name. The δ symbol marks the delta path for R.

4.10.2 Legend

We include a few useful definitions (in the form of a legend) to help users understand the hybrid tree diagram and the notations used in the code.

- $V_{rels}^{@v}[vars]$ defines view V at variable v in the view tree, over relations $rels$, and with free variables $vars$
- Every circle represents a variable
- Every rectangle represents a relation
- Every relation has its own color
- Double-circled nodes **must** be materialized
- Rectangles with rounded corners are streams (i.e. updatable relations)

Figure 4.10: Legend

4.10.3 Views' definitions

The last component of a PDF is the code section. This section contains all the code listings necessary in order to define any view or delta view. Originally, all pieces of code were valid M3 but we decided to change that and use our operators described in *Chapter 2: Preliminaries*. We believe that this leads to shorter but still comprehensive definitions while keeping the same notations we have in this thesis. This approach also protects our users in case something related to M3's syntax changes at some point in the future.

$$V_{R,S,U,T}^{@A} = \bigoplus_A((V_{U,T}^{@E}[A] \otimes V_{R,S}^{@B}[A]) \otimes \text{A})$$

$$\delta V_{R,S,U,T}^{@A} = \bigoplus_A((V_{U,T}^{@E}[A] \otimes \delta V_{R,S}^{@B}[A]) \otimes \text{A})$$

$$V_{R,S}^{@B} = \bigoplus_B(V_S^{@D}[A,B] \otimes V_R^{@C}[A,B])$$

$$\delta V_{R,S}^{@B} = \bigoplus_B(V_S^{@D}[A,B] \otimes \delta V_R^{@C}[A,B])$$

$$V_R^{@C} = \bigoplus_C \bigoplus_{extra} \text{R[A,B,C,extra]}$$

$$\delta V_R^{@C} = \bigoplus_C \bigoplus_{extra} \delta\text{R[A,B,C,extra]}$$

$$V_S^{@D} = \bigoplus_D \text{S[A,B,D]}$$

$$V_{U,T}^{@E} = \bigoplus_E(V_U^{@F}[E] \otimes \text{T[A,E]})$$

$$V_U^{@F} = \bigoplus_F \text{U[E,F]}$$

The definitions listed above correspond to a simple SQL query that joins together all relations and then aggregates over A.

4.11 M3 parser

We also provide another parser, this time for M3, which is used to check that our freshly generated code is syntactically correct. Like the SQL parser, this one is also based on the latest DBToaster M3 parser which is extended in order to provide support to custom types.

An optional check is performed right after the code generation stage: our M3 code is put through our custom M3 parser. The parser will detect and report if any syntax error or syntactically ambiguous constructs have been introduced at any stage during our pipeline. This way, only valid M3 code is passed to the DBToaster's back-end and we do not have to rely on any external code to catch this sort of errors.

Because parsing any file takes linear time in the size of the file, which can get huge for real-life schemas with hundreds of relations and thousands of variables, there is a flag to deactivate this process.

Chapter 5

Applications

Summary

In this chapter we show how our system can be used in much more complex scenarios by using task-specific rings.

5.0.1 The Housing dataset

Housing is a synthetic dataset emulating the textbook example for the house price market [Ng14]. It consists of six tables: `House`(postcode, price, number of bedrooms/-bathrooms/garages/parking lots, living room/kitchen area, etc.), `Shop`(postcode, opening hours, price range, brand, e.g., Costco, Tesco, Sainsbury's), `Institution`(postcode, type of educational institution, e.g., university or school, and number of students), `Restaurant`(postcode, opening hours, and price range), `Demographics`(postcode, average salary, rate of unemployment, criminality, and number of hospitals), and `Transport`(postcode, the number of bus lines, train stations and distance to the city centre for the postcode). The natural join of all relations is on the common attribute (*postcode*) and has 27 attributes (variables). For this schema, we consider an optimal view tree that has each root-to-leaf path consisting of query variables of one relation.
Note: Since the only shared and therefore relevant variable is *postcode* we give all other variables canonical names.
We get the following SQL database schema:

Listing 5.1: Housing.SQL

```
CREATE TABLE HOUSE(postcode int, h2 int, h3 int, h4 int, h5 int, h6 int, h7
    ↪ int, h8 int, h9 int, h10 int, h11 int)
FROM FILE './House.tbl' LINE DELIMITED CSV(delimiter := '|');


CREATE TABLE SHOP(postcode int, s2 int, s3 int, s4 int, s5 int, s6 int)
FROM FILE './Shop.tbl' LINE DELIMITED CSV(delimiter := '|');
```

```
CREATE TABLE INSTITUTION(postcode int, i2 int, i3 int)
FROM FILE './Institution.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE RESTAURANT(postcode int, r2 int, r3 int)
FROM FILE './Restaurant.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE DEMOGRAPHICS(postcode int, d2 int, d3 int, d4 int, d5 int)
FROM FILE './Demographics.tbl' LINE DELIMITED CSV(delimiter := '|');

CREATE TABLE TRANSPORT(postcode int, t2 int, t3 int, t4 int)
FROM FILE './Transport.tbl' LINE DELIMITED CSV(delimiter := '|');
```

We would like to answer two SQL queries over this dataset:

Listing 5.2: housing_query_count.SQL

```
SELECT COUNT(postcode)
FROM HOUSE NATURAL JOIN SHOP NATURAL JOIN INSTITUTION NATURAL JOIN RESTAURANT
    ↪ NATURAL JOIN DEMOGRAPHICS NATURAL JOIN TRANSPORT;
```

and

Listing 5.3: housing_query_sum.SQL

```
SELECT SUM(postcode)
FROM HOUSE NATURAL JOIN SHOP NATURAL JOIN INSTITUTION NATURAL JOIN RESTAURANT
    ↪ NATURAL JOIN DEMOGRAPHICS NATURAL JOIN TRANSPORT;
```

Since both queries are over the same join we can combine them into

Listing 5.4: housing_query.SQL

```
SELECT COUNT(postcode), SUM(postcode)
FROM HOUSE NATURAL JOIN SHOP NATURAL JOIN INSTITUTION NATURAL JOIN RESTAURANT
    ↪ NATURAL JOIN DEMOGRAPHICS NATURAL JOIN TRANSPORT;
```

This query can be solved by a simple DBMS using two traversals of the temporary table obtained from the join. The first traversal computes the count and the second, the sum. Modern DBMSs use highly advanced query optimizers so it is quite likely that there would be only one traversal. But would it be possible to ensure that only one traversal is used no matter what DBMS or query optimizer we use?

For our framework, the answer is yes! Because we rely on rings to model the structure of our answers, we can transform the two computations COUNT($postcode$) and SUM($postcode$) into only one SUM($f(postcode)$) where $f(postcode) = \langle 1, postcode \rangle$ (in other words, we do a sum over pairs containing both the count and the sum). Since

we only use one SUM we are certain that there will be only one traversal of our huge temporary table. But now, we have to provide a way for SQL to add or multiply two pairs together. In our system this is trivial, we simply provide DBToaster with a ring specification.

Listing 5.5: pair_ring.hpp

```
struct RingAvg{
    int count;
    int sum;

    static RingAvg zero;

    RingAvg(int c, int s): count(c), sum(s){}

    RingAvg(): count(0), sum(0){}

    RingAvg& operator += (const RingAvg &r){
        this->count += r.count;
        this->sum += r.sum;
        return *this;
    }
};

RingAvg RingAvg::zero = RingAvg(0, 0);

RingAvg operator * (const RingAvg &l, const RingAvg &r){
    return RingAvg(l.count * r.count, l.sum * r.count + r.sum * l.count);
}

RingAvg operator * (int alpha, const RingAvg &r){
    return RingAvg(alpha * r.count, alpha * r.sum);
}

RingAvg f(int postcode){
    return RingAvg(1, postcode);
}
```

5.0.2 Other applications

[NO18] presents three different use cases of F-IVM which we will briefly mention here:

- Matrix Chain Multiplication: Given a sequence of matrices A_1, A_2, ..., A_n, the goal is to find the most efficient way to multiply these matrices. We can model each matrix A_i as a relation R_i with the payloads carrying the actual matrix values. The optimal variable order corresponds to the optimal sequence of matrix multiplications that minimizes the overall multiplication cost, which is the textbook Matrix Chain Multiplication problem [CLRS09].
- Gradient computation used for learning linear regression models over joins [SOC16]
- Factorized Representation of Query Results: Our framework is flexible enough to support scenarios where the view payloads are themselves relations representing results of conjunctive queries, or even their factorized representations. We only need to define the relational data ring that allows us to store entire relations as payloads.

Chapter 6

Experiments

Summary

In this chapter we compare the code generated by our system (gen-F-IVM) against manually-optimized use cases of F-IVM (hw-F-IVM) as well as 1-IVM (first-order IVM) and DBT (DBToaster's fully recursive higher-order IVM). We also compare against two algorithms based on recomputing the results from scratch on every update.

Our experimental results can be summarized as follows:

- Both gen-F-IVM and hw-F-IVM outperform DBT on both datasets
- hw-F-IVM is slightly faster (10% on *Housing* and 0.07% on *Retailer*) than gen-F-IVM which was expected since it was coded using much more advanced optimization techniques
- IVM algorithms based on re-evaluation significantly underperform for evolving datasets

In the following sections we describe our experiments in more detail.

6.1 Benchmark Setup

All experiments have been performed on an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz with 24GB 2400MHz DDR4 RAM (ASUS ROG GL553VE) running Ubuntu 16.04 LTS. The C++ code has been compiled using gcc version 7.3.0 with the following flags:

- -std=c++14
- -pedantic

- -Ofast
- -Wall
- -Wextra

We use a modified version of DBToaster 2.2 in single-threaded mode.

6.2 Implementations

We compare six IVM algorithms:

- **gen-F-IVM** which uses the code our system generates
- **hw-F-IVM** which uses highly-optimized manually written code
- **1-IVM** which uses DBToaster's first-order IVM algorithm
- **DBT** which uses DBToaster's fully recursive higher-order IVM
- **gen-F-RE** which performs the re-evaluation using variable orders
- **DBT-RE** which performs the re-evaluation using DBToaster

All six algorithms are implemented on top of DBToaster's back-end but the first two use a slightly different back-end than the last four. We only modified the back-end to support custom rings so, for our experiments, we can consider the two back-ends equivalent.

6.3 Datasets

We run our experiments over two of the datasets introduced in [NO18]:

1. *Housing* is a synthetic dataset modeling a house price market. We also use this dataset in *Chapter 5: Applications*. It consists of six relations: `House`, `Shop`, `Institution`, `Restaurant`, `Demographics` and `Transport` arranged into a star schema and with 1.4M tuples of positive numbers in total. The natural join of all relations is on the common attribute (*postcode*) and has 27 attributes (variables). For this schema, we consider an optimal view tree in which each path from the root to a leaf consists of the variables of one relation. This dataset has been designed to analyze how redundant data in the join affects various IVM algorithms.

2. *Retailer* is a real-world dataset used by a retailer for predicting user demands and for decision support. The dataset has a snowflake schema with one relation `Inventory` with 84M records, storing information about the inventory unit for a product in a location, at a given date. The `Inventory` relation joins on three different dimensions: `Item` (on *product_id*), `Weather` (on *location* and *date*), and `Location` (on *location*) with `Census` (on *zipcode*). The natural join of these five relations has 43 attributes. For this schema, we consider a view tree in which the partial order on join variables is: *location* - { *date* - { *product_id* }, *zip* } and the variables of each relation form a distinct root-to-leaf path.

Note: We use the same raw data and variable orders that are also used in all experiments in [NO18].

6.4 Queries

We analyze the four algorithms in the case where we maintain a variable on top of a natural join. For the *Housing* dataset, we sum over the common join variable *postcode* and for the *Retailer* dataset we sum over the *inventoryunits* attribute of `Inventory`.

For *Housing* we use the following query:

Listing 6.1: housing_query.SQL

```
SELECT SUM(postcode)
FROM HOUSE NATURAL JOIN SHOP NATURAL JOIN INSTITUTION NATURAL JOIN RESTAURANT
    ↪ NATURAL JOIN DEMOGRAPHICS NATURAL JOIN TRANSPORT;
```

For *Retailer* we use the following query:

Listing 6.2: retailer_query.SQL

```
SELECT SUM(inventoryunits)
FROM INVENTORY NATURAL JOIN LOCATION NATURAL JOIN CENSUS NATURAL JOIN ITEM
    ↪ NATURAL JOIN WEATHER;
```

For both cases, we measure the average throughput (as tuples/second) for incrementally maintaining all relations under updates grouped in batches of size 1,000.

6.5 Results

	gen-F-IVM	hw-F-IVM	DBT	1-IVM	gen-F-RE	DBT-RE
Housing	24,827,041	26,015,095	19,718,310	2,865,162	51,421*	473*
Retailer	3,573,678	3,599,652	1,327,617	3,591,294	2,524*	2,383*

Figure 6.1: The average throughput (tuples/sec) of incrementally maintaining a sum aggregate under updates of size 1,000 to all relations of the *Housing* and *Retailer* datasets. * denotes a computation stopped after 20 minutes.

Figure 6.1 summarizes our results. The handwritten version of F-IVM achieves the highest average throughput in both cases. The generated version comes pretty close to the handwritten one, processing about 10% fewer tuples on the *Housing* dataset. That happens because the strategy used by hw-F-IVM uses a smarter multiplication which in the end saves a map update in the resulting code. Our system is not yet capable of taking advantage of the structure of the schema. The difference between them is insignificant on *Retailer*.

We can also conclude that re-evaluation strategies are not suitable for continuously evolving databases (even if we group updates together).

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis we have taken the state of the art algorithm F-IVM for dealing with the problem of incrementally maintaining an arbitrary query with joins, projections, and group-by aggregates, and turned it into a lightweight, full-scale Scala software system that together with a modified version of DBToaster's back-end is capable of producing highly-optimized C++ query engines based on F-IVM. We also built a graphical tool, embedded into our system, that can be used to create complex, aesthetically pleasing, brief but comprehensive models of our system's output (i.e. the optimized maintenance strategy, the high-level DBToaster trigger code generated for each view in the maintenance strategy as well as the declarative view definition). Furthermore, we showed that our system can deal with much more complex schemas and datasets for a wide variety of applications because of the use of task-specific rings. As a further matter, there are various ways in which we can improve our system, and we shall present a couple of ideas we discussed but did not have time to implement in the next section.

7.2 Future work

7.2.1 Single-tuple updates

We have implemented batch updates as delta relations, which we defined as a collection of tuples mapped to payloads. But we can also have single-tuple updates, which can be implemented much more efficiently due to their size and structure. We mention that DBToaster already supports single-tuple updates so this future extension to your system will bridge the gap of functionality between it and DBT.

7.2.2 Factorizable updates

F-IVM is also flexible enough to support factorizable updates, an alternative approach which does not represent updates as delta relations but as a union of factorizable relations. This idea can make our system much faster since the cumulative size of the decomposition can be much less than the size of the original delta relation.

7.2.3 Support for cyclic queries

[NO18] also mentions an IVM variant for cyclic queries which extends the view tree with indicator projections that identify the active domains of the relations [AKNR16]. Such projections have no effect on the query result but can constrain view definitions thus bringing asymptotic savings in both space and time.

7.2.4 Parallel computing

One could also modify the system to work on multiple CPU cores by using the fact that computing (delta) view trees is a highly parallelizable problem (i.e. two subtrees of the same node can be computed in parallel as they are data-independent). One can also notice that several stages of our pipeline are mutually independent (e.g. PDF generation and code generation). Therefore, we can place these processes into two separate threads of execution without having to worry about the order in which they work or finish.

7.2.5 More applications

We could also apply our framework to other scenarios. In order to do that, we do not have to change anything but the ring specification. Using more scenarios can be useful for testing and for proving F-IVM's versatility.

7.3 Reflections

I chose this project without knowing anything about factorized databases, IVM or almost any concept this thesis or [NO18] uses. I only chose this project because I enjoyed the second part of the 2^{nd} year course on Databases, which, in retrospect, was nothing more than simple SQL and a brief introduction into database management system (DBMS) implementation. I would never have guessed that in about 14 months I will be finishing my own system in Scala, a language I only used in coursework for Imperative programming or Object-oriented programming. I am glad I chose Scala for this project as pattern

matching, higher-order functions, case classes, the type system and the rich set of standard libraries were extremely useful and made a lot of things shorter and easier to code than it would have been in other object-oriented languages like C++ or Java.

The project has also introduced me to the Incremental View Maintenance world and I have learnt a lot while researching F-IVM related work. I have seen that it is incredibly hard to find a solution that performs optimally in all possible cases and that in general, there are no one-size-fits-all algorithms. However, F-IVM presents an ingenious approach, that of performing all computations over some carefully chosen ring instead of the original data, and I am thankful I have been the first one to implement a system based on it.

References

[AKNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.

[AtCG+15] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, et al. Design and Implementation of the LogicBlox System. In *SIGMOD*, pages 1371–1382, 2015.

[CG+14] Badrish Chandramouli, Jonathan Goldstein, et al. Trill: A high-performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4):401–412, 2014.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[CY12] Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends in DB*, 4(4):295–405, 2012.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.

[GOW15] Todd J. Green, Dan Olteanu, and Geoffrey Washburn. Live programming in the LogicBlox system: A MetaLogiQL approach. *PVLDB*, 8(12):1782–1791, 2015.

[KAK+14] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014.

[Koc10] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, pages 87–98, 2010.

[Ng14] Andrew Ng. CS229 Lecture Notes. Stanford & Coursera, `http://cs229.stanford.edu/`, 2014.

[NO18] M Nikolic and DA Olteanu. Incremental view maintenance with triple lock factorisation benefits. Association for Computing Machinery, 2018.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[Ora] Materialized View Concepts and Architecture. `http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmview.htm`.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, pages 3–18, 2016.

[SQL] Create Indexed Views. `http://msdn.microsoft.com/en-us/library/ms191432.aspx`.