DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases

Yanif Ahmad and Christoph Koch
Department of Computer Science
Cornell University, Ithaca, NY
{yanif, koch}@cs.cornell.edu

ABSTRACT

We present DBToaster, a novel query compilation framework for producing high performance compiled query executors that incrementally and continuously answer standing aggregate queries using in-memory views. DBToaster targets applications that require efficient main-memory processing of standing queries (*views*) fed by high-volume data streams, *recursively* compiling view maintenance (VM) queries into simple C++ functions for evaluating database updates (*deltas*). While today's VM algorithms consider the impact of single deltas on view queries to produce maintenance queries, we recursively consider deltas of maintenance queries and compile to thoroughly transform queries into code. Recursive compilation successively elides certain scans and joins, and eliminates significant query plan interpreter overheads.

In this demonstration, we walk through our compilation algorithm, and show the significant performance advantages of our compiled executors over other query processors. We are able to demonstrate 1-3 orders of magnitude improvements in processing times for a financial application and a data warehouse loading application, both implemented across a wide range of database systems, including PostgreSQL, HSQLDB, a commercial DBMS 'A', the Stanford STREAM engine, and a commercial stream processor 'B'.

1. INTRODUCTION

Static query workloads are commonly posed on relational data management systems, in the form of view declaration queries, repetitive (parameterized) queries from client-side application logic, and continuous queries for stream processing. However, in today's data management systems, these queries are answered using the same machinery as for flexible, interactive query processing, namely query plan interpreters and other runtime components. While many database systems include a compiler that produces and optimizes query plans, we argue that this model of compilation does not push the envelope far enough. We propose DBToaster, a novel approach for compiling SQL aggregate queries into extremely efficient C++ code for continuous standing query evaluation.

DBToaster is a SQL query compilation framework for main memory databases that produces C++ code to incrementally maintain aggregate views at high update rates using aggressive delta processing techniques. Our work is motivated by applications that require highly efficient answering of fixed aggregate query workloads, as in data stream processing, online data warehouse loading, and in financial applications. We focus on main-memory databases due to the prevalence of standing queries in stream processors, where compilation has only recently been considered [3, 6].

DBToaster works by *recursively* compiling queries into incremental view maintenance code; that is, while data increments for queries are traditionally expressed and evaluated again as queries, we recursively compute increments to these increments, and so forth. Recursive compilation provides three key advantages. First, our C++ code processes query plan execution paths, eliminating overheads in interpreting query plans stored in dynamic data structures. Next, we generate asymptotically simpler code at each recurrence, since computing increments allows us to avoid certain database scans or joins. Finally, we tailor code generation to produce native code, enabling modern C++ compilers to apply aggressive inlining and other optimizations, resulting in compact straight-line code sequences.

We showcase DBToaster in a few applications that are served in limited fashion by today's data management tools, including algorithmic order book trading (algos), and an integrated approach to data warehouse loading and analysis. Our compiled query processors are several orders of magnitude faster than state-of-the-art databases and significantly outperform stream processing engines on such workloads. In the case of queries on order book data for algos, our approach stands alone in its ability to support realistic data rates without resorting to very substantial computing clusters. Indeed, the memory consumption of our main-memory techniques is sufficiently low to support applications such as data warehouse loading.

2. DBTOASTER OVERVIEW

Data and Query Model. DBToaster focuses on applications issuing standing queries on a database which subsequently process continuously changing, large volumes of input tuples. DBToaster is capable of compiling a wide variety of SQL queries including the core relational algebra, standard aggregates (`sum`, `avg`, `count`, `min`, `max`), subqueries

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

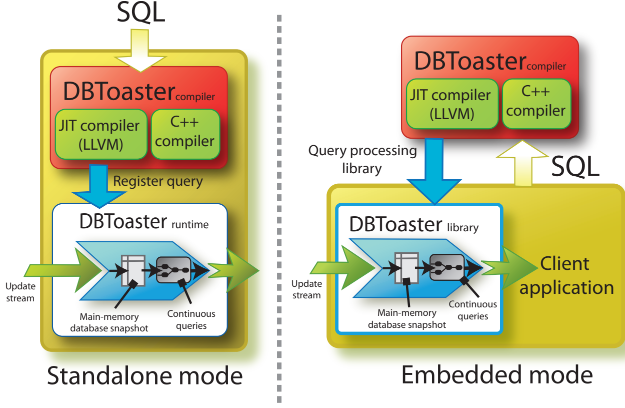

Figure 1: DBToaster architecture, illustrating standalone and embedded modes of query processing. In standalone mode, DBToaster produces a lightweight runtime, while in embedded mode, DBToaster can alternatively produce a binary library that can be used directly in client applications.

and nested aggregates. Our data model differs from today's data stream processors, in that we consider a database as a set of relations each subject to an arbitrary sequence of inserts, updates and deletes. In contrast, data stream processors assume a well-defined separation between tuples' insert and delete operations on the stream (value- or count-based windows), or assume ordered deletion semantics (punctuations or heartbeats). In DBToaster, each tuple has an arbitrary lifetime, thus our standing queries process a database spanning an arbitrary valid time using temporal database terminology. Indeed, many stream applications, such as order book trading and moving object applications, are self-managing, in that the application logic and delta patterns ensure that state does not grow unboundedly in practice.

System Model. At its core, DBToaster consists of a parser, an algebraic compiler and a code generator. Our compilation workflow produces a delta-processing function for each type of delta (insert, update or delete) on any base relation used in the query. In addition, compilation defines in-memory aggregate views that are maintained during runtime to support our delta-processing functions. We will see how these data structures are defined in the description of our algorithm below. The DBToaster runtime may be used as a standalone query processor accepting input over a network interface or archived stream, or as an embeddable query processor that can be directly compiled into the same address space as application logic. DBToaster also exposes a read-only interface to its internal data structures to support ad-hoc client-side queries. DBToaster includes a debugger and profiler for tracing delta processing functions and their maintenance of internal data structures.

3. QUERY COMPILATION

We now present DBToaster's compilation algorithm through an example illustrating how queries are turned into efficient procedural code. Our compilation framework applies to the core relational algebra and group-by aggregates, and uses a custom query algebra to define map data structures. These maps are closely related to views definable by SQL aggregate group-by queries but at the same time are main memory data structures that are easy to access in applications. Due to space limitations, we present a small fraction of our map algebra simplifications as needed for our example query. Our full map algebra is approximately 70 simplification rules.

Compilation example. Consider the query below on three relations and schemas $R(A,B), S(B,C), T(C,D)$:

```
select sum(A*D) from R, S, T
    where R.B=S.B and S.C=T.C
```

Given the data and query model above, we assume relations R, S, T are manipulated via update streams which consist of the standard requests of inserting, updating and deleting tuples. For ease of presentation, we can consider updates as pairs of delete and insert requests. We start with handling an insert to the relation R, with a tuple $\{\langle a,b\rangle\}$. Assuming the variable q maintains the query result, we can show:

Insert R(a,b):

$$\begin{aligned}\Delta q &= \mathrm{sum}_{A*D}(\{\langle a,b\rangle\} \bowtie S \bowtie T)\\ &= \mathrm{sum}_{A*D}(\{a\} \times \sigma_{B=b}(S) \bowtie T)\\ &= \mathrm{sum}_{a*D}(\sigma_{B=b}(S) \bowtie T)\\ &= a * \underbrace{\mathrm{sum}_{D}(\sigma_{B=b}(S) \bowtie T)}_{q_D[b]}\end{aligned}$$

Above $q_D[b]$ is an example of a map that we use to compute the change in query result q, a map with key-value entries of keys b, and values defined as the result of the query: $\mathrm{sum}_D(\sigma_{B=b}(S) \bowtie T)$. While we do not go into the full details of the derivation validity, we can see it is a simplification of the original query by considering the relation R as a singleton relation $\{\langle a,b\rangle\}$. We can symmetrically derive for inserting into relation T as: $\Delta q = d * q_A[c]$, resulting in a map $q_A[c] = \mathrm{sum}_A(R \bowtie \sigma_{C=c}(S))$. An insert on S is:

Insert S(b,c):

$$\begin{aligned}\Delta s &= \mathrm{sum}_{A*D}(R \bowtie \{\langle b,c\rangle\} \bowtie T)\\ &= \mathrm{sum}_{A*D}(\sigma_{B=b}(R) \times \sigma_{C=c}(T))\\ &= \underbrace{\mathrm{sum}_A(\sigma_{B=b}(R))}_{q_A[b]} * \underbrace{\mathrm{sum}_D(\sigma_{C=c}(T))}_{q_D[c]}\end{aligned}$$

Note the elimination of any join in the above query since we are able to exploit distributivity properties of summation and multiplication, and the cross product operator. At this point we have presented one level of compilation, for an insertion into each base relation R, S, T, resulting in incremental query result computation code, a set of maps which we have to maintain, and queries defining the map contents. At this point, we recursively compile the map definition queries, considering each of the three types of insertion (to R, S, T). We subsequently aggressively inline any code generated into a handler for each type of insertion. For example, consider the maps $q_A[c], q_A[b]$ above, whose entries are dependent on the relation R. We recursively compile incremental maintenance of this map for insertions to R as:

Insert R(a,b):

$$\begin{aligned}\Delta q_A[b] &= \mathrm{sum}_A(\{\langle a,b\rangle\}) = a\\ \text{foreach } c\text{: } \Delta q_A[c] &= \mathrm{sum}_A(\{\langle a,b\rangle\} \bowtie \sigma_{C=c}(S))\\ &= \mathrm{sum}_a(\sigma_{BC=bc}(S))\\ &= a * \underbrace{\mathrm{sum}_1(\sigma_{BC=bc}(S))}_{q_1[b,c]}\end{aligned}$$

Recursion level	Event	Query, Q, to compile	Code for ΔQ	Maps used in code	Map definition
1	$\pm R$	$\text{sum}_{A*D}(R \bowtie S \bowtie T)$	$\pm a * q_D[b]$	$q_D[b]$	$\text{sum}_D(\sigma_{B=b}(S) \bowtie T)$
1	$\pm S$	$\text{sum}_{A*D}(R \bowtie S \bowtie T)$	$\pm q_A[b] * q_D[c]$	$q_A[b]$ $q_D[c]$	$\text{sum}_A(\sigma_{B=b}(R))$ $\text{sum}_D(\sigma_{C=c}(T))$
1	$\pm T$	$\text{sum}_{A*D}(R \bowtie S \bowtie T)$	$\pm d * q_A[c]$	$q_A[c]$	$\text{sum}_A(R \bowtie \sigma_{C=c}(S))$
2 2	$\pm R$ $\pm R$	$\text{sum}_A(R \bowtie \sigma_{C=c}(S))$ $\text{sum}_A(\sigma_{B=b}(R))$	foreach(c): $\pm a * q_1[b,c]$ $\pm a$	$q_1[b,c]$ (no new maps)	$\text{sum}_1(\sigma_{BC=bc}(S))$
2 2	$\pm S$ $\pm S$	$\text{sum}_A(R \bowtie \sigma_{C=c}(S))$ $\text{sum}_D(\sigma_{B=b}(S) \bowtie T)$	$\pm q_A[b]$ $\pm q_D[c]$	(no new maps) (no new maps)	
2 2	$\pm T$ $\pm T$	$\text{sum}_D(\sigma_{B=b}(S) \bowtie T)$ $\text{sum}_D(\sigma_{B=b}(T))$	foreach(b): $\pm d * q_1[b,c]$ $\pm d$	$q_1[b,c]$ (no new maps)	$\text{sum}_1(\sigma_{BC=bc}(S))$
3	$\pm S$	$\text{sum}_1(\sigma_{BC=bc}(S))$	± 1	(no new maps)	

Figure 2: DBToaster's recursive compilation of the 'select sum(a*d) from R, S, T' query, showing the query being compiled, the procedural code required to incrementally compute the query result, maps required by the code, and the query defining the map. Above, the event $\pm R$ indicates both an insert and delete on R, and we present the code in one piece, although DBToaster would produce different event handlers.

Above, we use sum_1 to refer to a count aggregate, that is $\text{sum}_1(\sigma_{BC=bc}(S))$ is a count of $\langle b, c\rangle$ tuples in S. Note the *foreach* statement when computing $\Delta q_A[c]$. This arises since a single tuple $\langle a, b\rangle$ in R affects all map entries with keys c^* where the relation S contains tuples $\langle b, c^*\rangle$. Again our compilation is symmetric for the relations R and T due to the nature of the join graph in this query. Thus the maintenance code for maps $q_D[b]$, and $q_D[c]$ is:

Insert T(c,d):

$$\begin{aligned}\Delta q_D[c] &= d\\ \Delta q_D[b] &= \text{foreach(c): } d * q_1[b,c]\end{aligned}$$

For an insertion to S, we must maintain maps $q_A[c], q_D[b]$:

Insert S(b,c):

$$\begin{aligned}\Delta q_A[c] &= \text{sum}_A(R \bowtie \{\langle b,c\rangle\})\\ &= \text{sum}_A(\sigma_{B=b}(R) \times \{c\})\\ &= \text{sum}_A(\sigma_{B=b}(R)) \;=:\; q_A[b]\\ \Delta q_D[b] &= \text{sum}_D(\{\langle b,c\rangle\} \bowtie T)\\ &= \text{sum}_D(\{b\} \times \sigma_{C=c}(T))\\ &= \text{sum}_D(\sigma_{C=c}(T)) \;=:\; q_D[c]\end{aligned}$$

Note that we are already maintaining maps $q_A[b], q_D[c]$ above, that is, we can exploit map sharing opportunities across event handler functions. Finally, we maintain $q_1[b,c]$ for insertions to S simply as: $\Delta q_1[b,c] = 1$. We show the resulting handler functions from this example, including the inlining of code fragments generated at each recursive step.

```
// Declarations
result q;
map q_A_b, q_A_c, q_D_b, q_D_c, q_1_bc;

// Event handlers
void on_insert_into_R (attribute a, attribute b) {
   q += a * q_D_b[b];   q_A_b[b] += a;
   foreach (c in q_A_c.keys()) do
      q_A_c[c] += a * q_1_bc[b][c];
}

void on_insert_into_S (attribute b, attribute c) {
   q += q_A_b[b] * q_D_c[c];   q_A_c[c] += q_A_b[b];
   q_D_b[b] += q_D_c[c];       q_1_bc[b][c] += 1;
}

void on_insert_into_T (attribute c, attribute d) {
   q += q_A_c[c] * d;   q_D_c[c] += d;
   foreach (b in q_D_b[b].keys()) do
      q_D_b[b] += q_1_bc[b][c] * d;
}
```

Additionally Table 2 compactly describes this compilation example, including the case of deletion events which turn out to be strictly analogous in this case due to the fact that sum aggregates have a well defined inverse as subtraction.

4. DEMONSTRATION SETUP

The DBToaster demonstration presents the map algebra, the compilation workflow, and the performance advantages of compiled query processors over alternative database architectures. In this section we describe the application scenarios that act as motivating use cases for DBToaster, as well as the visualization tools that convey the technical aspects of query transformations and compiled executor performance. Since DBToaster is suited for applications exhibiting high volume update streams, in this demonstration, we show DBToaster processing queries for an automated trading application making use of NASDAQ TotalView order book data [2], and emulating a combined data warehouse loading and analysis application for TPC-H data.

Processing order books in equities trading. Order books provide a superior view of the market microstructure for use in trading algorithms. The bid order book consists of prices and volumes at which investors are willing to buy equities, and correspondingly the ask order book indicates investors' selling orders. Investors continually add, modify or withdraw limit orders, thus we regard order books as relations subject to high volumes of order deltas. Note order books do not grow unboundedly in practice, but cannot be expressed by windows given arbitrary input deltas. We present a few queries in the automated trading application, first a volume-weighted average price (VWAP) query which computes the average price-volume product of orders making up a given fraction of volume in the bid and ask order books. One example usage of the VWAP metric is for a static order book imbalance (SOBI) trading strategy, which detects trade price movements based on whether there is greater activity in the bids or asks order book. The final query detects strategies being employed by market makers

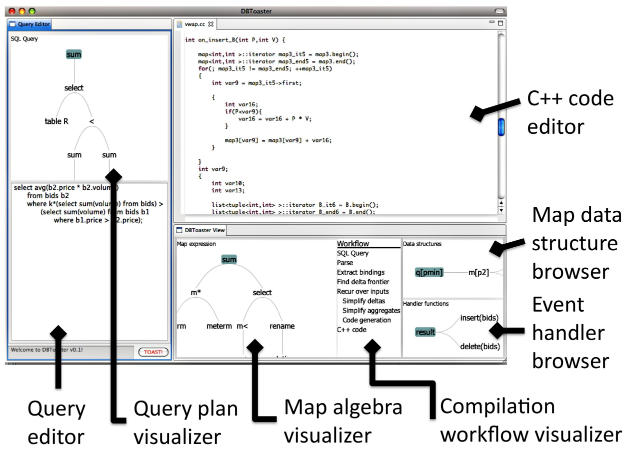

Figure 3: DBToaster compilation process visualization, displaying map algebra transformations, generated code, and internal views maintained.

through the order book, where market makers often submit orders to entice buyers or sellers into the market to aid in balancing their position.

Data warehouse loading. Loading large data warehouses is a computation-intensive process, hence most data warehouse loading is performed offline. While commercial warehouse loaders use highly tuned code for aggregation, incoming data is often the result of costly, inefficient data integration queries, which often blow up data sizes to cause inefficient loading. Compiling data integration and aggregation queries together yields efficient code for loading the warehouse and may avoid the materialization of large intermediate results. We use DBToaster to jointly process loading a warehouse from an OLTP database, and an aggregation query on the warehouse. We emulate the data integration step by using a data cleaning query to convert a TPC-H dataset into a star schema from the Star Schema Benchmark (SSB) [5]. We then evaluate query 4.1 from SSB on the transformed TPC-H dataset.

Interactive demonstration. An integral part of this demo is to support interaction with conference attendees, thus in addition to providing canned queries implementing these applications, we allow attendees to directly pose their own queries on the TotalView and TPC-H datasets.

4.1 Query compilation and code generation

The first of our two visualization tools, Figure. 3, conveys the compilation process to demo attendees. This tool visually displays a standard relational query plan, and illustrates the compiler workflow in a step-by-step fashion, including map algebra simplifications and the maps instantiated during compilation. We place particular emphasis on the recursive nature of our compilation, demonstrating compilation of deltas on the queries corresponding to our map data structures. At this point query compilation is complete, and we use a pair of browser windows listing both the maps and the event handling functions generated to aid in discussions with attendees. We also use a debugging tool to provide step-by-step tracing of map maintenance when processing a delta.

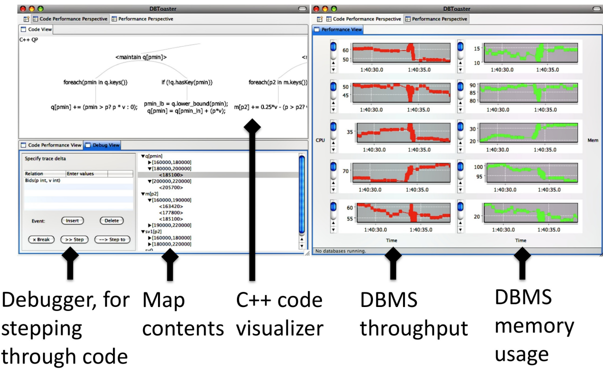

Figure 4: DBToaster debugger supporting stepping and tracing query processing and map maintenance, and performance visualizer for comparing against alternative databases in the DBMS bakeoff.

4.2 DBToaster vs. DBMS* Bakeoff

This demo also presents DBToaster's competitiveness with a variety of database tools, by performing a DBMS bakeoff. Our comparison points are PostgreSQL, a pure Java main-memory DBMS (HSQLDB [1]), a commercial DBMS 'A', the Stanford STREAM engine [4], and a commerical stream processor 'B'. We have a visualization tool (Figure. 4) to show the performance achieved by the each database system including tuple throughput, memory usage and cache performance. We also present detailed profiling of DBToaster's compiled code breaking down its overheads for each map, the binary size, and finally the compile time including both the C++ generation and the subsequent compilation to a native binary. To provide an entertaining audience experience, we run an audience challenge to find queries both yielding the greatest performance over the other database engines in the bakeoff, as well as queries that illustrate the poorest performance. Attendees will be provided with two laptops at the demonstration booth to experiment with queries, and encourage participation by displaying a leaderboard of the running results.

5. REFERENCES

[1] HSQLDB, http://www.hsqldb.org.

[2] NASDAQ TotalView order book, http://www.nasdaqtrader.com/Trader.aspx?id=TotalView.

[3] B. Gedik, H. Andrade, K.-L. Wu, P. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *Proc. of the 2008 ACM SIGMOD*, pages 1123–1134, 2008.

[4] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial CIDR*, Jan. 2003.

[5] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF , 2007.

[6] J. Salz and R. Tibbetts. StreamBase Systems. Stream processor with compiled programs, U.S. Patent Application, app. #11/644,189, pub. #US 2008/0134158 A1. Filed December 2006.