Holistic Data Access Optimization for Analytics Reports*

ABSTRACT

Object-Relational Mappers (ORMs) enable single language access to both the main memory data and the database data of an application. Unfortunately, they also lead to performance inefficiencies, especially in analytics applications with information-rich reports involving nested and aggregated results over large data volumes.

Past database research suggests that a report over a database can be modeled by a single semi-structured query, i.e. a query involving nesting and heterogeneity. Thus, the report's data accesses can be holistically optimized through a single query. Collage resolves important practical challenges towards enabling the report-as-query approach and its holistic optimization. In particular, the Collage middleware system models a report page as a SQL++ query. SQL++ is a superset of SQL by two extensions: (a) It provides distributed access to an SQL database and the in-memory objects of the web application programming language. (b) The query output has the full generality needed by report templating engines built around HTML and JSON. Indeed, the SQL++ data model is a superset of JSON. The Collage implementation features its own report templating engine but one can easily utilize Collage's query processor in other HTML and JSON-based templating engines.

On the query optimization side, Collage expands into analytics queries. The following query optimization problems emerge and are solved. (1) Novel rewritings guarantee the compilation of any SQL++ query into an efficient set-at-a-time query plan, assuming the objects of the application programming language offer object id's that can be compared and be efficiently grouped. This guarantee includes analytics queries, i.e., queries that involve aggregation and top-k processing, which were unaddressed by prior works on semi-structured queries. (2) Normalized and denormalized set-at-a-time query plans are formulated and evaluated. The evaluation validates the manyfold speedup over a mainstream ORM. It also shows that normalized-sets plans outperform denormalized-sets plans by manyfold for the case of analytic reports, which was not the case in semistructured queries without aggregation.

The Collage query processor has been integrated and deployed as part of a complete web application development framework, including a templating enegine. Using an online IDE, one can build a wide class of web applications with both the ease of single language access, and performance of holistic optimization.

*Supported by NSF III-1018961 and NSF III-1219263, PI'd by Prof Papakonstantinou who is a shareholder of App2You Inc, which commercializes outcomes of this research.

1. INTRODUCTION

It has been well-known since the 80s that the *impedance mismatch* between the SQL database and the application layer is a major time waste in application development [8]. A developer has to use two languages (SQL and an application programming language, such as Java, Ruby or Python) for data access, and tediously convert between relational tables and objects.

Best practices among practitioners to mitigate impedance mismatch are to utilize *Object-Relational Mappers* (ORMs). With ORMs, developers specify mappings from relational tables into classes of the application programming language. Consequently, the ORM automatically translates navigation across objects of these classes into SQL queries, often involving joins. With this object navigation paradigm, ORMs offer *single language access* to all data, be them native objects of the application or tuples of the database. The pervasive adoption of ORMs[1], such as Hibernate, Active Record of Ruby-on-Rails (RoR) and CakePHP is a testimony to the importance of the impedance mismatch problem, and the desire for single language access to database tuples and application objects.

While ORMs provide single language unified access for simple data access patterns, this ease of use often comes at a performance cost of queries in very common reporting applications, especially analytics applications that display nested, aggregated data [18, 17]. As a running example, consider the web application in Figure 1 that has a dashboard page for visualizing and analyzing sales data of a TPC-H [22] database. A user uses the checkboxes to select nations to analyze, and the application stores the selections as in-memory objects within the HTTP session of the application server. The application then displays a HTML table of selected nations, where each row comprises the nation name

[1]In this paper, code examples are presented in RoR, since RoR is a widely used web framework, and Active Record, its built-in ORM, supports many mainstream database engines. Nevertheless, the discussion applies to other popular ORMs.

and the top 3 years of sales revenue.

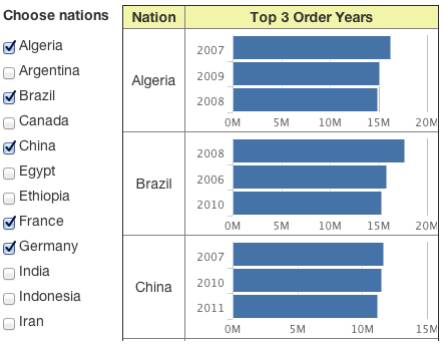


Figure 1: Analytics application for TPC-H data

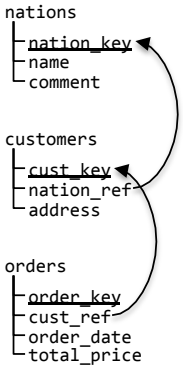


Figure 2: TPC-H schema

An ORM induces developers to *decompose* data access into multiple object navigations (translated by the ORM into multiple queries) that are interspersed among loops and conditionals of imperative languages. This decomposition reduces data access performance: Even though each individual query may be optimally executed by the database, the set of queries collectively may not compute the report efficiently. Such inefficiencies can be very large in analytics applications producing reports that involve nesting, joins and aggregation. In particular, this paper focuses on the inefficiencies of *tuple-at-a-time* queries, where each outer context tuple causes the execution of an inner query that produces nested tuples.

```
1   <!-- Template for checkboxes -->
2   <table>
3     <% Nation.all.each.do |n| %>
4       <% if session['selected_nations'][n.nation_key] == true then %>
5         <tr>
6           <td><%= n.name %></td>
7           <td>
8             <%  aggregates = Order
9                   .select('date_part(\'year\', order_date) as order_year,
10                           sum(total_price) as sum_price')
11                  .joins('customer')
12                  .where('nation_ref = ?', n.nation_key)
13                  .group('date_part(\'year\', order_date) as order_year')
14                  .order('sum_price DESC')
15                  .limit(3)  %>
16            <!-- Template for bar chart JavaScript component -->
17          </td>
18        </tr>
19      <% end %>
20    <% end %>
21  </table>
```

Figure 3: RoR page with tuple-at-a-time queries

For example, consider the RoR fragment in Figure 3 that implements Figure 1, and operates over the TPC-H schema in Figure 2. Data access code in the figure are denoted with italics. Line 3 uses an `each` loop with variable `n` to iterate through all nations, and Line 4 retains only nations that are selected by the checkboxes.

Within the loop and conditional, Line 6 produces the nation name, whereas Lines 7-17 produce the bar chart. Since arbitrarily complex statements can occur within the loop and conditional, RoR does not attempt to holistically optimize multiple data access points into more efficient queries. Instead, it simply executes statements in an imperative, tuple-at-a-time fashion: For each selected `Nation` object `n`, Lines 8-15 are translated into a SQL query that has a single parameter for the nation key of `n`. Parameterizing the inner queries causes k nations to require k queries, which almost always incurs a manyfold performance penalty due to extraneous sequential scans or repeated random accesses, as will be demonstrated in our experiments. Worse still, the performance penalty ratio increases with the amount of data accessed and visualized, since the number of queries issued is dependent on the number of nations.

A sophisticated developer who optimizes for performance will re-implement tuple-at-a-time queries as *set-at-a-time* queries. Two alternatives demonstrated in this paper are:

- **Denormalized-sets queries**: Queries that retrieve results as denormalized-sets. For example, a single query that retrieves the join of nations and top sales.
- **Normalized-sets queries**: Queries that retrieve results as normalized-sets. For example, a query that retrieves nations, and a separate query that retrieves top sales with corresponding nation ids.

```
1   # Copy selected nation keys from memory to database temporary tables
2   ActiveRecord::Base.connection.execute(
3     'CREATE TEMPORARY TABLE selected_nations (nation_ref INTEGER)' )
4   selected_nations.each do |n|
5     ActiveRecord::Base.connection.execute(
6       "INSERT INTO selected_nations(nation_ref) VALUES ('#{n}')" )
7   end

8   # Store common sub-expression for selected nations in temporary tables
9   ActiveRecord::Base.connection.execute(<<SQL
10    CREATE TEMPORARY TABLE temp_nations AS
11    SELECT n.nation_key, n.name
12    FROM   nations AS n
13    JOIN   selected_nations AS s ON s.nation_ref = n.nation_key
14  SQL
15  )

16  # Sub-query to retrieve distinct nation keys
17  q1 = 'SELECT DISTINCT nation_key AS nation_key_2 FROM temp_nations'

18  # Sub-query to retrieve aggregates en-masse
19  q2 = Arel::Table.new(:customers)
20    .project('t1.nation_key_2,
21              date_part(\'year\', o.order_date) as order_year,
22              SUM(o.total_price) AS sum_price')
23    .from('customers as c')
24    .joins('JOIN orders AS o ON c.cust_key = o.cust_ref ' +
25           "JOIN (#{q1}) AS t1 ON c.nation_ref = t1.nation_key_2")
26    .group('t1.nation_key_2,
27            date_part(\'year\', o.order_date)')

28  # Sub-query to partition aggregates by nation key
29  q3 = <<SQL
30    SELECT t2.*, row_number() OVER(
31                   PARTITION BY t2.nation_key_2
32                   ORDER BY t2.sum_price
33                 ) AS row_id
34    FROM (#{q2.to_sql()}) AS t2
35  SQL

36  # Execute query that retrieves top-3 aggregates in each partition
37  top_sales = Order
38    .select('t3.nation_key_2, t3.order_year, t3.sum_price')
39    .from("(#{q3}) as t3")
40    .where('row_id <= 3')

41  # Nest aggregates in respective nation keys
42  result = Hash.new { |hash, key| hash[key] = {
43      :name       => nil,
44      :agg_orders => []
45  } }
46  top_sales.each do |t|
47      result[t.nation_key_2].aggregates << {
48          :order_year => t.order_year,
49          :sum_price  => t.sum_price
50      }
51  end

52  # Join nation keys with nation names
53  nations = ActiveRecord::Base
54            .select('nation_key , name').from('temp_nations')
55  nations.each do |n|
56      result[n.nation_key].name = n.name
57  end
```

Figure 4: RoR page with normalized-sets queries

For example, consider the RoR fragment for normalized-sets queries in Figure 4, which is manyfold faster than the

tuple-at-a-time queries in Figure 3: our experiments show a 7x speedup on a 3GB TPC-H database.

While ORMs make it easy to write tuple-at-a-time queries, they do not facilitate the significantly more efficient set-at-a-time queries. A developer has to be a SQL expert to understand that, since the nested query computes aggregates and top-k results, retrieving en masse the top 3 sales years for each nation id requires partitions and window functions (`OVER (PARTITION BY ... ORDER BY ...)` in Lines 16-40), which is an advanced and esoteric feature in SQL 2003.

Worse, the developer is burdened with explicit distributed programming: he has to (a) copy intermediate results across the memory/database boundary while minimizing communication costs (Lines 2-7), (b) store common sub-expressions in temporary tables (Lines 9-15), (c) write ad-hoc imperative code to perform nesting and joins in the application layer (Lines 42-51, 53-57) and (d) utilize different APIs based on which source the data resides in (SQL commands in Lines 1-15, ActiveRecord in Lines 16-40, object APIs in Lines 41-57). Not only does this lack of *location transparency* fall short of single language access, it may also result in performance inefficiencies since neither the programming language nor the database can take advantage of optimization opportunities in the integration code across the two systems.

To address these issues, the database community has proposed many novel solutions that enable holistic optimization of queries in reporting pages. Systems such as FERRY [18] and SWITCH [17] compile imperative data access code into an algebraic representation, thereafter apply holistic optimization of nested queries. Separately, in declarative web frameworks such as Strudel [11] and Hilda [24], pages are modeled as nested queries, which is a first step towards holistically optimization of the page.

In the same spirit, we present Collage [2, 3, 4], a declarative framework for data-driven web applications that provide novel holistic optimizations of reporting queries in analytics applications. In the current implementation, a developer declaratively specifies the reported data with a page query, which executes over multiple data sources. The page query is written in SQL++, which is backwards compatible with SQL, captures the JSON data model, and easily integrates the application programming language objects that are accessed by reports. Furthermore, the page query is *distributed* over a persistent SQL database and in-memory application-level objects of requests and sessions.

In its current implementation, Collage simultaneously provides ease of use for the developer through the SQL++ single language access, as well as efficient performance through holistic query optimization within its distributed query processor. Thus, data access within analytics applications is reduced to a special semi-structured and distributed query processing problem.

Systems that compile imperative code into queries (like FERRY and SWITCH) also stand to benefit from using Collage's query processor, as it provides a flexible data model (both in terms of input and ouput) and novel optimizations for complex reports that involve aggregation and topk.

1.1 Contributions

This paper focuses on the distributed query processor of Collage, which builds upon existing techniques to introduce novel query processing optimizations:

Optimizing tuple-at-a-time queries We show that all SQL++ queries are *set processable*, and Collage utilizes novel rewritings to holistically optimize naive and inefficient tuple-at-a-time plans to efficient set-at-a-time plans (i.e. denormalized-sets or normalized-sets plans). To the best of our knowledge, these rewritings target the largest class of nested queries among that considered by prior work: They (a) capture aggregation and ranking, which are expensive and common in modern analytics applications, (b) are applicable over multiple levels of nesting (c) handle arbitrary types of correlation between the enclosing query and the nested query. Our experiments validate that the rewritings achieve a manyfold speedup (e.g. 7x on a 3GB TPC-H database). Furthermore, larger databases and heavier visualizations result in even greater speedups. We also observe that set-at-a-time plans outperform tuple-at-a-time plans by a large margin for heavy queries, and remain competitive for inexpensive ones, which suggests that the rewritings should be applied whenever possible.

Performance comparison of different optimizations Our performance analysis and extensive experiments reveal that rewriting to normalized-sets plans is highly superior to denormalized-sets plans, especially for complex analytics queries. The efficiency gains are attributable to both obvious reasons, such as data redundancies and communication overhead incurred by denormalization, as well as surprising ones, such as SQL database optimizations enabled through the removal of outer-joins.

Lightweight integration of external objects Using Collage as middleware, a developer can easily use SQL++ as the single language to access and integrate objects of external data sources, including SQL databases, JSON data and Java objects. These objects can be defined in a different programming language and type system (thus going beyond object databases), and in contrast to conventional data integration systems, the developer does not incur the upfront burden of mapping external classes into SQL++ schemas. This is enabled by a novel SQL++ feature where query paths navigate lazily into external objects, and each path step dynamically bind to the next object reference returned by a method call. Collage treats external objects as first-class citizens, and extends query processing correspondingly for heterogeneity and lenient type checking.

We have deployed Collage as a complete web application framework made available on the cloud. A demo system, which includes multiple sample queries such as the running example of Figure 1, is online at `http://ec2-54-244-248-12.us-west-2.compute.amazonaws.com`[2]. Furthermore, Collage is currently deployed in real-world applications, including an analytics application currently utilized by two pharmaceutical companies.

This paper is structured as follows. Section 2 illustrates the syntax and semantics of SQL++ through an example Collage application. Sections 3 and 4 present the technical details of Collage, including the SQL++ data model, SQL++ language extensions and optimizations for tuple-at-a-time queries. Section 5 compares the performance gains of Collage's optimizations. Finally, Section 6 presents related work in database and programming language research.

[2]The machine is hosted anonymously in the AWS cloud service in compliance with SIGMOD's double-blind requirements.

2. ARCHITECTURE, SYNTAX AND SEMANTICS

Collage provides an efficient *template engine* [23] for web frameworks that follow the Model-View-Controller (MVC) architecture pattern, which is practically all web frameworks. In MVC web frameworks, an application is modularized into actions (Controllers) and pages (Views). A HTTP request invokes an action, which is implemented in the native language of the web framework (e.g. Ruby for RoR, Java for Spring Web). The action reads and writes the application state, which includes a persistent SQL database and transient in-memory objects stored in request and session scopes of the application server. An action terminates by choosing a page to be displayed next. To display the chosen page, the web framework then uses a template engine to evaluate a page template which, in adherence to MVC best practices, should simply instantiate HTML markup without causing side-effects on the application state. Web frameworks allow different template engines to be installed as plugins. For example, RoR works with template engines such as ERB (as shown in Figure 3), Haml, and Slim.

Collage's template engine can be used in Java-based[3] MVC web frameworks, such as Struts and Spring Web. Collage is also available as a standalone web framework, which further mitigates impedance mismatch in actions by specifying them as PL/SQL statements [4]. However, in this paper we focus on Collage's template engine for pages that are analytics reports, and do not further discuss the use of Collage for actions.

```
1   <table>
2     <fstmt:for query="
3       SELECT    n.nation_key, n.name
4       FROM      session.selected_nations AS s,
5                 db.nations AS n
6       WHERE     s.nation_ref = n.nation_key AND
7                 s.selected = true
8     ">
9       <tr>
10        <td> { name } </td>
11        <td>
12          <funit:bar_chart>
13            <fstmt:for query="
14              SELECT    db.date_part('year', o.order_date) AS order_year,
15                        SUM(o.total_price) AS sum_price
16              FROM      db.orders AS o,
17                        db.customers AS c
18              WHERE     o.cust_ref = c.cust_key AND
19                        c.nation_ref = nation_key
20              GROUP BY  db.date_part('year', o.order_date)
21              ORDER BY  sum_price
22              LIMIT     3
23            ">
24              <column>
25                <label> { order_year } </label>
26                <value> { sum_price } </value>
27              </column>
28            </fstmt:for>
29          </funit:bar_chart>
30        </td>
31      </tr>
32    </fstmt:for>
33  </table>
```

Figure 5: Collage page that inlines queries within markup

In Collage, page templates (such as Figures 5 and 6) assemble the data of the page with SQL++ queries, which are distributed over the database and in-memory objects. The semi-structured query results are rendered via Collage's markup into HTML and JavaScript components, such as maps, charts, and calendars. Therefore, report pages are easily specified using only semi-structured distributed queries and template markup.

[3]The same fundamental principles can be used to integrate Collage's template engine with web frameworks of other languages such as Ruby or PHP.

```
1   <fstmt:with target="shown_nations">
2     SELECT  n.nation_key, n.name, (
3               SELECT   db.date_part('year', o.order_date) AS order_year,
4                        SUM(o.total_price) AS sum_price
5               FROM     db.orders AS o,
6                        db.customers AS c
7               WHERE    o.cust_ref = c.cust_key AND
8                        c.nation_ref = nation_key
9               GROUP BY db.date_part('year', o.order_date)
10              ORDER BY sum_price
11              LIMIT    3
12            ) as aggregates
13    FROM    session.selected_nations AS s,
14            db.nations AS n
15    WHERE   s.nation_ref = n.nation_key AND
16            s.selected = true
17  </fstmt:with>

18  <table>
19    <fstmt:for source="shown_nations">
20      <tr>
21        <td> { name } </td>
22        <td>
23          <funit:bar_chart>
24            <fstmt:for source="aggregates">
25              <column>
26                <label> { order_year } </label>
27                <value> { sum_price } </value>
28              </column>
29            </fstmt:for>
30          </funit:bar_chart>
31        </td>
32      </tr>
33    </fstmt:for>
34  </table>
```

Figure 6: Collage page that splits queries from markup

As with all template engines, the developer can combine queries and markup by either *inlining* queries within the markup (e.g. Figure 5, which is similarly inlined as the RoR page of Figure 3) or *modularizing* them into two separate parts (e.g. Figure 6). Typically inlining is preferred for simple pages, whereas modularizing is for pages with complex data access and visualizations. In Figure 5, the semantics of the `fstmt:for` statement (Lines 2-8) is that it evaluates its query, and instantiates its body (Lines 9-31) for each tuple in the result. For example, a `tr` element is output for each nation. Expressions enclosed in "`{`" and "`}`" utilize attributes of the enclosing query. Expressions within HTML tags are output as strings (Line 10), whereas those within visual unit tags such as `funit:bar_chart` (Lines 25-26) are used as the model of JavaScript components. Collage is responsible for efficiently and incrementally rendering both HTML and JavaScript components [2].

Collage accepts SQL++ queries, which are a superset of SQL queries and thus familiar to a large audience of SQL developers. Sections 3 and 4 provide details on these SQL++ extensions. Internally within the Collage framework, a *page compiler* always compiles the `fstmt:for` queries of a page into a single SQL++ query with nesting, as in Figure 6, so that Collage can subsequently optimize the SQL++ query by rewriting the initial tuple-at-a-time plan into a more efficient set-at-a-time plan (Section 4.2).

3. DATA MODEL

The input to the SQL++ queries is essentially a graph of tuples. Formally, an *SQL++ object* is a pair of an *id* and an *SQL++ value*. A *named* id can be explicitly written in queries. An *unnamed* id cannot be explicitly written in queries. An *SQL++ value* is either

1. a typed *atomic value*, such as the string `'abc'` or the integer `7`.

2. an *object reference*, which is either the id of an object or the special constant `null`.
3. a *tuple* $[a_1 : v_1, \ldots, a_n : v_n]$, where the attribute names $a_1, \ldots, a_n$ are strings and each attribute value v_i is an SQL++ value.
4. a *collection* $\{e_1, \ldots, e_n\}$, where each e_i is recursively an SQL++ value.

Since SQL++ is an extension of SQL, Collage easily models an SQL database as an SQL++ source. Given a standard SQL relation `R`, Collage models the SQL database as an object identified by the named id `R`. The value of `R` is a collection of tuple values, where the tuples have atomic values.

Objects of programming languages (Java, Javascript, etc) are also easily wrapped into SQL++. In the running example, there is a session attribute `selected_nations` whose value is a Java `List` of (references to) objects that have a boolean property `selected` and an integer property `nation_ref`. Collage models this session attribute and the data accessible via it as an object identified by the name `selected_nations`. The value of `selected_nations` is a collection of object references to objects that have an unnamed id and a tuple value, where one of the attributes is named `selected` and has boolean values.

DAG and cyclic structures are possible by having object references that point to the same object. Notice that this modeling is virtual, in the sense that no activity will take place until a query needs to reach some of the reachable data. Therefore, even if a huge amount of Java data are accessible by navigations that follow object references, none will be accessed until a query needs to.

Notice that SQL++ can also be seen as an extension of JSON. JSON is particularly important because it is readily consumed by the visualization components utilized in Collage's template engine and other template engines.

4. QUERY LANGUAGE

The SQL++ query language is backwards-compatible with SQL, featuring extensions analogous to the extensions that the SQL++ data model introduced over the SQL data model:

Nested value output Unlike SQL which restricts the `SELECT` clause to only allow sub-queries producing scalars, in SQL++ the `SELECT` clause can contain any SQL++ subquery. This extension enables creating nested values. For example, the running example's query (Figure 6, Lines 2-16) creates a collection of tuples, where each tuple corresponds to a nation and has an attribute `aggregates` whose value is itself a collection of tuples.

The default `SELECT-FROM` query creates collections of tuples - as in SQL. Minor syntactic variations of the `SELECT` clause enable the creation of other types of collections, such as collections of atomic values and collections of collections. In any of their many incarnations, nested collections present opportunity for performance optimization via the set-based processing techniques described in this paper.

Heterogeneity In SQL the `FROM` clause introduces (tuple) variables that range over the tuples of the relations. The SQL++ `FROM` clause extends SQL's `FROM`, by allowing variables to also range over any type of member that a collection has.

Unlike SQL whose `UNION` requires both arguments to be homogeneous (i.e. identical schema), SQL++ allows the arguments to be heterogeneous (i.e. different schemas), therefore being able to creating output heterogeneous results even when the input is homogenous.

Navigation in nested values and navigation through object graphs In the spirit of XQuery and OQL, the SQL++ `FROM` allows clauses where a variable ranges over the collection that is bound to another variable, potentially involving navigation through tuples. The navigations of SQL++ can access nested values and can also access the nesting that is created by object references, without making a distinction between the two.

Location transparency A developer writes a SQL++ query in a location-transparent fashion, even when it spans multiple data sources. In the running example, the developer has mapped (not shown) the `db` data source to a SQL database, and the `session` data source to the in-memory HTTP session object of the application server. In Figure 6, database tables (Lines 5, 6, 14) and in-memory objects (Line 13) are prefixed by respective data source names. *Source-specific functions* (such as the `db.date_part` stored procedure, Line 3) are similarly prefixed, whereas *flexible functions* (such as `=` in Lines 7, 13, 14) that can be executed at any data source are not prefixed.

Non-set results Unlike SQL where queries always output tables, SQL++ queries can be simple expressions. For example, `1 + 1` outputs a SQL++ scalar. This paper does not further discuss such queries, since they do not present set processing opportunities.

4.1 Plans and Algebra Overview

A logical plan $p = T_1 \leftarrow e_1; \ldots; T_n \leftarrow e_n; e$ starts with a list of zero or more assignments $T_i \leftarrow e_i$, where each T_i is a *temporary result* and each e_i is a SQL++ algebra expression that may input the previously computed temporaries $T_1, \ldots, T_{i-1}$. The result of p is the SQL++ table resulting from the evaluation of the *result expression* e, which may input any temporary $T_1, \ldots, T_n$.

The algebraic operators involved in SQL++ algebra expressions input and output tables, i.e., collections of homogenous tuples. Intuitively, each attribute of a table corresponds to a variable of the `FROM` clause or some value that is reachable via the variables. Each tuple corresponds to a binding of the variables. Therefore we will call them *binding attributes* and *binding tuples* respectively.

The majority of SQL++ algebra operators are extensions of operators well-known from conventional SQL processing (e.g., see textbooks [15]). While conventional operators input and output tables where the values of the binding attributes are only atomic values, SQL++ extends the operators to allow the binding attribute values to also be collections, tuples and object references. The list of standard operators comprises cartesian product $\times$, union $\cup$, intersection $\cap$, difference $-$, selection σ_c, join $\bowtie_c$, full outer-join $\rtimes\ltimes_c$, left outer-join $\rtimes_c$, semi-join $\ltimes_c$, anti-semi-join $\overline{\ltimes}_c$, projection $\pi_{\bar{L}}$, sort $\tau_{\bar{F}}$ (where $\bar{F}$ is the order-by list of terms, which initially appear in the SQL++ `ORDER BY` clause), top-k Topk_k (which returns the first k tuples of its input), duplicate elimination δ, group-by $\gamma_{\bar{G};f_1 \to N_1,\ldots,f_m \to N_m}$ ($\bar{G}$ is the terms that appear in the `GROUP BY` clause and $f_1, \ldots, f_m$ are the aggregate functions).

In the remainder, given a binding tuple t of a bindings table, an attribute name a that is not already the name of a binding attribute of t and a value v, the notation $t\#[a : v]$ denotes the tuple that results from adding the attribute/value pair $a : v$ on the list of attribute/value pairs that t already

has.

The list of novel SQL++ operators, and their uses, are as follows:

Translating the `FROM` clause with Ground, Scan and Navigate operators The role of the *ground* (Ground) and *scan* (Scan) operators is to provide the algebraic counterpart of the `FROM` clause. The ground operator has no input table (indeed it is the only operator with no input table) and always returns $\{[]\}$, i.e., a table that has a single tuple with no binding attribute.

The *scan* operator $\text{Scan}_{s\mapsto a}$ inputs a table. For each binding tuple t of the input table, the *scan* finds the collection $v_c = \{e_1, \ldots, e_n\}$ identified by s,[4] where either (a) s is a named object id (e.g., `customers`, `selected_nations`), in which case v_c is the value of the named object, or (b) s is a binding attribute of t and v_c is its value. In either case, the *scan* outputs binding tuples $t\#[a : e_1], \ldots, t\#[a : e_n]$.

The *navigate* operator ($\text{Nav}_{s.p_1.\ldots.p_m\mapsto a}$) outputs exactly one binding tuple $t\#[a : v]$ for each input binding tuple t. The value v is non-null, when the binding tuple t has an attribute s whose value is a tuple t_1 or an object reference pointing to a tuple t_1. In either case, the tuple t_1 has an attribute p_1 whose value is a tuple t_2 with an attribute p_2. The navigation continues until a tuple t_m is reached and t_m has an attribute p_m whose value is v. If there is no such path of tuples $t_1, \ldots, t_m$ with attributes $p_1, \ldots, p_m$, the value v is `null`. In the case that the value of s is a collection with a single tuple (or single reference to a tuple), it is treated as if it were a tuple or an object reference.

For notational brevity, this paper also uses the derived operator *scan-navigate* (ScNa) that combines a *scan* operator that produces a binding attribute a, with multiple *navigate* operators that initiate their navigation from a. In particular, $\text{ScNa}_{s\mapsto a;a.p^1\mapsto a^1,\ldots,a.p^n\mapsto a^n} \equiv$ $\text{Nav}_{a.p^1\mapsto a^1} \ldots \text{Nav}_{a.p^n\mapsto a^n} \text{Scan}_{s\mapsto a}$. Furthermore, the output attribute name can be omitted if it is identical to the input attribute name.

Tuple-at-a-time nesting operator Nested queries are translated into SQL++ algebra using the *apply-plan* operator $\alpha_{P\to N}$. The α operator inputs a single table. In general, P is a correlated plan, such that attributes of the input table may appear in P wherever constants or named id's can appear in uncorrelated queries. Such correlated input attributes are called *parameters* of P. The α operator outputs one tuple $t\#[N : P/t]$ for each input tuple t, i.e., each output tuple has all attributes of the respective input tuple t, and an additional attribute N for the result of evaluating the plan P/t, i.e. the non-correlated plan that results from substituting each parameter with its respective binding in the tuple t.

Nesting via grouping SQL++ allows aggregate functions that produce non-scalar results. Of particular interest to the optimizations presented in this paper is the $\texttt{NEST}(A_1, \ldots, A_t) \to N$ function whose output is a new table N. Formally, the function is evaluated on each input sub-collection $I_1, \ldots, I_l$ created by $\gamma_{G_1,\ldots,G_n;\texttt{NEST}(A_1,\ldots,A_t)\to N}$ according to the group-by attributes $G_1 \ldots G_n$. For each input sub-collection I_i, the function `NEST` performs the projection $\pi_{A_1,\ldots,A_n}(I_i)$ and outputs the table attribute N.

Partition The partition operator χ enables support of the SQL 2003 `PARTITION` feature and window functions. In addition, χ is important for optimizing aggregation queries: Section 4.2 shows that even if the SQL++ query has no `PARTITION`, the apply-plan rewriter may introduce a χ. Formally, $\chi_{G_1\ldots G_n;e_{\mathbf{X}}}$ operates on an input table that has attributes $G_1 \ldots G_n$. It groups the input tuples into input sub-collections $I_1, \ldots, I_l$ according to the group-by attributes $G_1 \ldots G_n$, such that each input sub-collection I_i has an associated group-by tuple $(G_1 : g_{1_i}, \ldots, G_n : g_{n_i})$. For each input sub-collection I_i, the operator derives an output sub-collection O_i by replacing the parameter input $\mathbf{X}$ with I_i in the expression $e_{\mathbf{X}}$. The operator's result is $\cup_{i=1,\ldots,l}\{(g_{1_i}, \ldots, g_{n_i}) \times O_i\}$, i.e., the union of the output sub-collections after they are padded with their group-by values. For example:

$\chi_{\texttt{nation};\text{Topk}_2(\tau_{\texttt{sales}}(\mathbf{X}))}$

nation	name	sales
USA	Joe	6
China	Chen	5
China	Fu	8
China	Zhao	4

$=$

nation	name	sales
USA	Joe	6
China	Fu	8
China	Chen	5

Functions SQL++ allows functions over input attributes in the `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `GROUP BY`, `JOIN`, `PARTITION` and `LIMIT` clauses. A *flexible* function can be executed in both memory and database sources, whereas a *source-specific* function can only be executed in one source. For example, `=` is a flexible function, whereas `db.date_part()` (Figure 6, Line 3) is a database-specific function.

In the final distributed plan, each operator is designated to execute in a specific source. To handle the case where a `SELECT`, `WHERE`, `ORDER BY` or `LIMIT` clause involves both memory-specific and database-specific functions, the SQL++ operators $\pi_{\bar{L}}$, σ_c, $\tau_{\bar{F}}$ and Topk_k are restricted from naming functions in their parameters. Rather, the function operator $\lambda_{f(A_1,\ldots,A_n)\to N}$ evaluates a single function f. For each input tuple t with attributes $A_1, \ldots, A_n$, the λ operator outputs an output tuple that has all the input attributes, and an additional attribute N that is the result of evaluating f using the $A_1, \ldots, A_n$ values in t. For example, the SQL++ clause `ORDER BY g(X),f(Y,Z)` is translated into: $\tau_{N_1,N_2}(\lambda_{f(Y,Z)\to N_2}(\lambda_{g(X)\to N_1}))$.

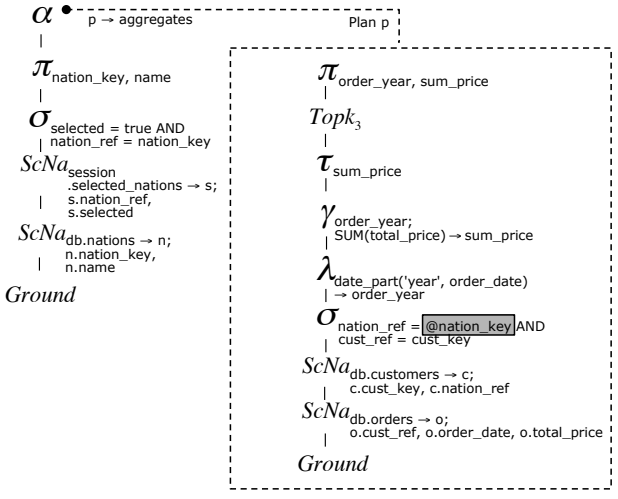


Figure 7: Initial plan from translation into algebra

Figure 7 shows the initial algebraic plan translated from the SQL++ query of Figure 6. The $Scan_{\texttt{session.selected\_nations}}$ operator scans an in-memory Java collection object and outputs a tuple with attributes `nation_ref` and `selected`. The

[4] In case s identifies a non-collection value v and coercion is enabled, the case is treated as if s had identified the collection $\{v\}$.

α operator represents the `SELECT` clause nested query: the correlated plan p (within the dotted box) corresponds to Lines 3-11 of the SQL++ query. Notice the use of the parameter `@nation_key` in the correlated plan. According to the semantics of α, for each nation input tuple t, the correlated plan is evaluated with the parameter `@nation_key` substituted by the corresponding value in t. The set of tuples resulting from the correlated plan evaluation becomes the value of the new attribute `aggregates`. The function `date_part` is evaluated in a λ operator.

4.2 Apply-plan Rewriter

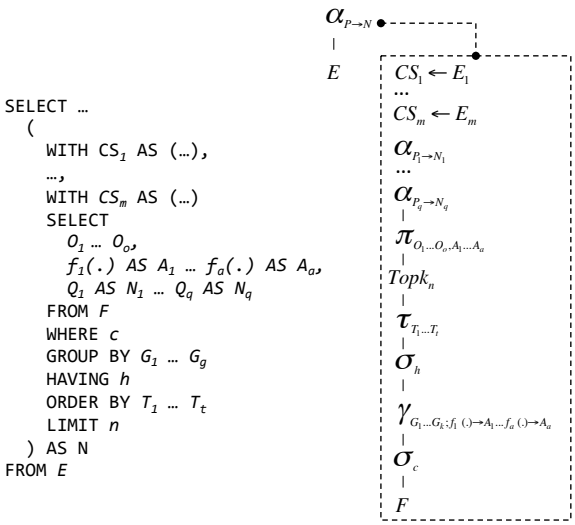


Figure 8: Pattern for Set-Processable Queries and Plans

The apply-plan rewriter inputs a plan (eg, see the pattern of Figure 8) and replaces tuple-at-a-time α operators with set-at-a-time operators, thereby producing plans as the one shown in Figure 9.

The following case analysis proceeds in the following steps of increasing complexity:

1. We first present the special case where the `FROM` clause has no join or outerjoin expressions (called *expression-free FROM* case) and the plans of the α operators either do not involve or any aggregation or they involve both a group-by clause and aggregation (called *no-total-aggregation* case). The analysis will reveal the potential requirements placed on the object references and object id's of the data.

2. Then we extend to `FROM` clauses that may involve expressions. This case encompasses the expression-free-FROM case.

3. Finally, we present the *total aggregation* case where we remove α operators whose plans involve aggregation without grouping. For example, a total aggregation case would be a modification of the running example where the nested query involves no grouping and returns the total sum of sales for its nation parameter. The total aggregation case has a fundamentally different rewriting from the no-total-aggregation case.

4.3 Expression-free FROM with no total aggregation

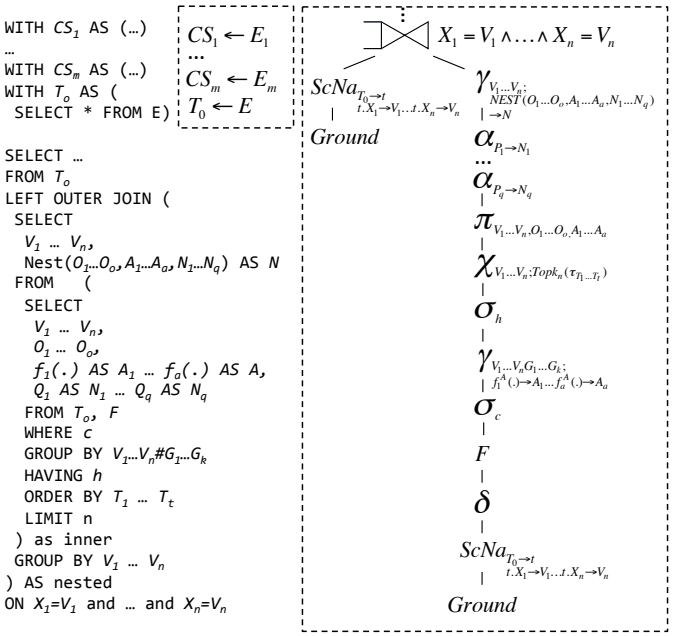


Figure 9: Pattern output by apply-plan rewrite rule

Suppose attributes $X_1, \ldots, X_n$ of expression E (i.e. the input of α) are the parameters of the correlated plan P. The rewriter replaces the α with the plan of Figure 9, which intuitively proceeds in the following steps.[5]

First the input E of the α is evaluated and assigned to a temporary T. In the running example, this means that nation keys and names are computed. See in Figure 10, which shows the rewritten plan of Figure 7, that the temporary `T1` contains the nation keys and names. Notice that the nation key would have to be computed even if it were not included in the `SELECT` clause of the outer query. The reason is that the nation key is necessarily an input of α since it is a parameter of its nested plan.

Second, the subplan of the right hand side of the outerjoin of Figure 9 is computed. This subplan has one output tuple $t = [X_1 : v_1, \ldots, X_n : v_n, N : P/(X_1 : v_1, \ldots, X_n : v_n)$ for each distinct tuple of parameters $v_1, \ldots, v_n$. This tuple also contains a binding attribute N, which carries the result $P/(X_1 : v_1, \ldots, X_n : v_n)$, i.e., the result of the execution of the nested plan P when its parameters are instantiated to $v_1, \ldots, v_n$. IN the running example, the right hand side subplan has attributes `nation_key` and `aggregates`.

Notice that the subplan has emerged from rewriting the ground of the original nested plan with a duplicate-eliminating (δ) projection of the parameters in E. Conceptually, this guarantees that the subplan will produce all combinations of parameters with tuples from the `FROM` clause of the nested plan P. (Again, we stress that one should think of this explanaion purely "conceptually", since subsequent optimizations will introduce various optimization efficiencies.)

A number of issues, pertaining to aggregation and top-k, have to be taken care of by the rewriting. In particular, if there were any grouping (γ) in the original plan, then its grouping attributes have to also include the parameters. If there were any ordering (τ) and top-K (Topk) they have

[5]Notice that the plan does not yet describe the fine details of its execution, neither is optimized for execution yet. These optimizations and details will be added later.

to be replaced by a partition (χ) operator. In the running example, the top-3 sales dates computation is achieved by partitioning the input by the `nation_key` parameter and then keeping the top-3 of each partition.

Finally, the natural left outer-join $\rtimes$ combines by joining on the parameters, the temporary T (which intuitively carries the result of the outer query, along with parameters) with the right hand side subplan (which carries the result of the inner query, also along with parameters). Simplifications of the rewrite rule are easy to produce for when one or more of the assignments, π, Topk, τ, σ or γ operators are absent.

In the case that the plan P of the α contained any assignments, these assignments would simply be set-processed (by application of the exact same rewritings that we use to rewrite the result expression) and would be computed before the computation of the outerjoin.

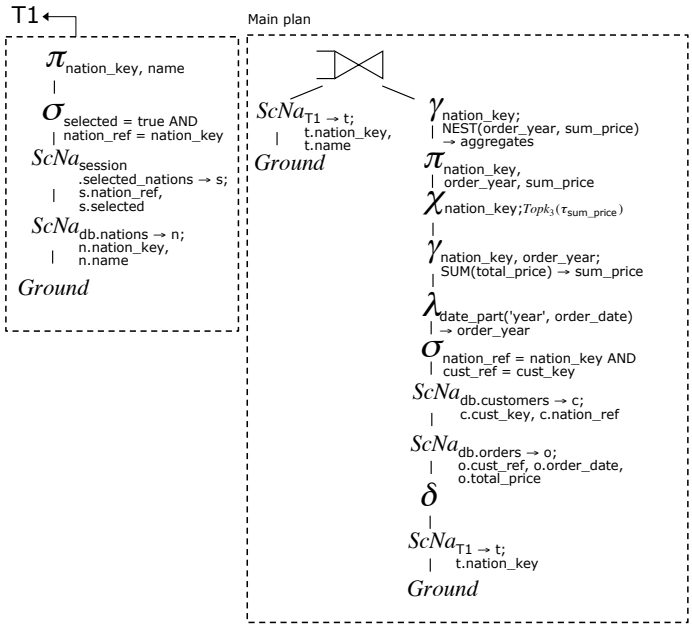


Figure 10: Plan after apply-plan rewriting

We call this type of plan a *normalized-sets plan*, i.e. for $\gamma_{G_1,\ldots,G_n;\texttt{NEST}}$, the grouping attributes includes only output attributes from E that are in correlated attributes $X_1, \ldots, X_n$. For example in Figure 10, the $\gamma_{\texttt{NEST}}$ has only grouping attribute `nation_key` which corresponds to the correlated attribute, whereas the non-correlated `name` is deferred until after the $\rtimes$. Section 4.8 explains why these plans are superior than alternate *denormalized-sets* plans where the grouping attributes of $\gamma_{\texttt{NEST}}$ also includes non-correlated attributes, i.e. the input of $\gamma_{\texttt{NEST}}$ is denormalized with respect to the correlated attributes.

When there are more than one α operator in a plan, the rewriter eliminates them by repeated application of the rewrite rule in a top-down manner: an α that appears in a nested plan is eliminated *after* its parent α has been eliminated.

Requirements of set processability on object id's and references Under certain circumstances described next, the reduction of tuple-at-a-time processing to set-at-a-time processing (set processability) is dependent on the availability of object references and object-id's at the source data. In particular, consider a nested `FROM` clause "`FROM` ⟨collection parameter⟩ v", whereas the variable v ranges over the members of a collection that is provided as a parameter from the outer level. The algebraic counterpart will be a plan of the form $\alpha_{P \mapsto N} E$, where the nested plan P is $\text{Scan}_{c \mapsto v}$ Ground and the outer plan E produces a binding attribute c. Note that we deliberately focus on the *scan* corresponding to the nested query's `FROM`. It will be obvious how the point below applies even in the presence of selections, groupings, joins, additional scans etc.

The elimination of α in this example requires that the parameter c appears in the group-by list. From an implementation point of view, this is efficient (and therefore worthy of reducing tuple-at-a-time to set-at-a-time) only if c is a directly or indirectly identifiable object. We call it directly identifiable if c is an object. We call it indirectly identifiable if it is only the collection value of an attribute of a tuple that is an object (i.e., identifiable). In the indirect case, the tuple's id can effectively operate as a proxy for the missing id of c itself.

In either case, the source has to either explicitly support the export of id's to Collage, so that the grouping according to c is executed at Collage, or, the source needs to be able to efficiently group according to object references/id's, so that Collage can defer to the source. The Java runtime is such a source, in the sense that it enables an implementation where Collage can group by object references.

As an example of the above case consider the following query

```
SELECT c.name, (SELECT n.* FROM c.nations n) FROM session.c
```

This query is set processable only if `c.nations` is an object (direct case) or c is an object (indirect case).

4.4 The total aggregation case

An extra challenge is introduced by nested queries that feature aggregation but do not feature grouping. For example, consider the following query that computes the sum of sales per customer.

```
SELECT cust_key, (
   SELECT SUM(o.total_price) AS sum_price
   FROM db.orders o
   WHERE c.cust_key = o.cust_ref
FROM db.customers c
```

Then consider a customer c that has made no order yet. By following the provided α-removal rule we derive a plan where the sum of sales is `null`, as opposed to the correct, which is zero. In such cases where there are no grouping attributes, any null that appears in lieu of the value of the aggregate f should be replaced with the result of this function f when it is given an empty group We omit the algebraic specifics of performing this change of value.

4.5 Unrestricted FROM clause

Next we consider `FROM` clauses that contain join and/or outerjoin expressions. While joins can be rewritten away, outerjoins cannot be rewritten away and therefore we will focus on them. We will first consider the case where the `FROM` clause of the nested query is solely a single outerjoin. In particular, consider the expression $\alpha_{l \bar{\bowtie}_c r \mapsto N} E$, where the left hand side l of the outerjoin involves parameters $\bar{x_l}$ and the right hand side r of the outerjoin involves parameters $\bar{x_r}$.

If the right hand side r is uncorrelated, i.e., $\bar{x_r} = \emptyset$, then the α removal proceeds similarly to the expression-free FROM case. Namely, the temporary $T = \delta\pi_{bar x_l, \bar{x_r}} E$ is computed first. Then the ground of l is replaced with T (the resulting expression denoted as $l(T)$) and the usual combination of $\gamma_{\texttt{NEST}}$ and outerjoin accomplishes the required nesting. In summary, the rewritten plan is

$$T \mapsto \delta\pi_{bar x_l, \bar{x_r}} E;$$
$$E \rtimes \gamma_{\bar{x_l};\texttt{NEST}(\bar{O})}(l(T) \rtimes_c r)$$

where O is the list of binding attributes of l and r.

If the right hand side r is also correlated (i.e., $\bar{x_r} \neq \emptyset$) then the ground of r is also replaced with T. Note however that in order to avoid cardinality errors in the result, a tuple of $l(T)$ should match a tuple of $r(T)$ only if they are derived from the same tuple of T. Technically this intuition is captured by changing accordingly the condition of the $\rtimes$. In particular, let us denote by r' the plan $r/(\bar{x_r} \mapsto \bar{x_r'})$, i.e., the plan that results from the substitution of the parameters $\bar{x_r}$ with parameters with fresh names $\bar{x_r}'$. Then the needed rewriting is

$$T \mapsto \delta\pi_{bar x_l, \bar{x_r}} E;$$
$$E \rtimes \gamma_{\bar{x_l};\texttt{NEST}(\bar{O})}(l(T) \rtimes_{\bar{x_r}=\bar{x_r'} \wedge c} r'(T))$$

It is straightforward to see how the above generalizes to the case where the plan of α were not simply an outerjoin but also included grouping, top-k etc.

4.6 Distributor

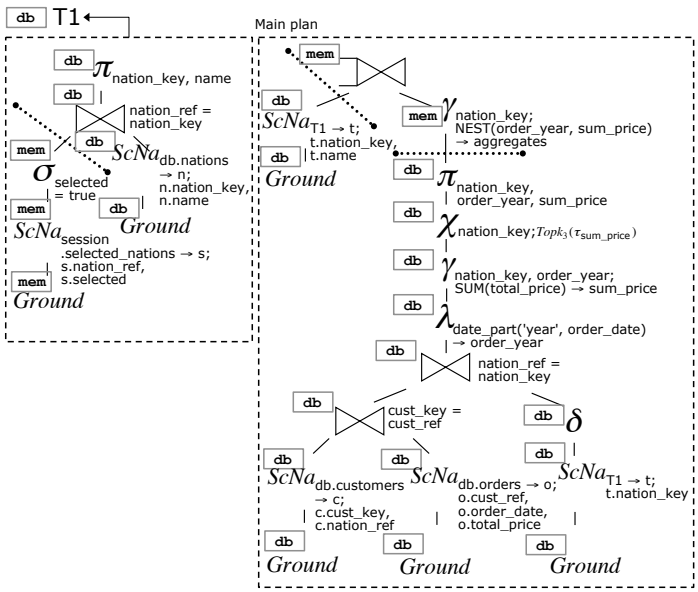


Figure 11: Plan after distributor's source decisions

The distributor decides the source where each operator will be executed, with the objective of outputting efficient physical plans that limit the number of tuples transferred between sources. Operators are either source-specific or flexible. A source-specific operator is: (1) a *ScNa* (2) an α (3) a λ_f where f is a source-specific function, or (4) a $\gamma_{\bar{G};f_1 \to N_1,\ldots,f_m \to N_m}$ where one or more of $f_1 \ldots f_m$ are source-specific. All other operators are flexible. For each operator, the distributor provides a *source decision* indicating whether it is executed in-memory or in-database.

The distributor is tuned for characteristics common to most web applications: persistent data in the database data are orders of magnitude larger than both (1) the transient memory data input by the SQL++ query, and (2) the output of the SQL++ query as displayed on the page. Therefore, the source decision of an operator is determined through bottom-up *sub-plan maximization* as follows: (i) If an operator is source-specific, it is decided to be executed in that source. (ii) If an n-ary operator (i.e. join, $\cup$ and $\cap$, which are all flexible) has children distributed across both sources, the operator is decided to be in-database, and the in-memory inputs are copied to the database. This is because copying in-memory inputs to the database is much faster than the opposite. (iii) Otherwise, the operator is flexible and has children within a single source, therefore the operator is decided to be in the same source as its children, in order to recursively maximize the size of sub-plans.

Figure 11 shows the plan after the distributor has determined source decisions, as denoted with `db` or `mem`. Cuts (i.e. data transfers) between a pair of adjacent in-memory and in-database operators are denoted with dotted perpendicular lines.

4.7 Plan-to-SQL Translation

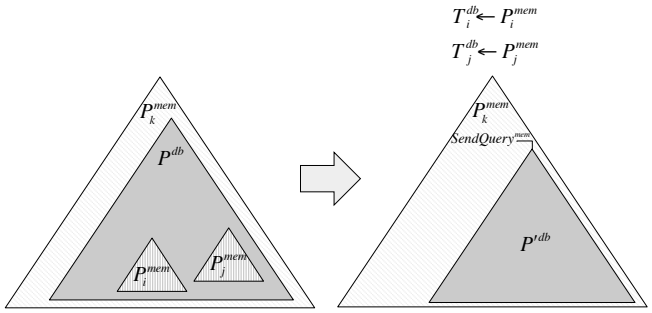


Figure 12: Translating Maximal Sub-plans to SQL

Figure 12 illustrates how the plan-to-SQL translator translates maximal sub-plans to SQL. Given source decisions on each operator, the plan-to-SQL translator performs the following recursively: Each maximal in-database sub-plan $P^{\texttt{db}}$ is replaced by a $SendQuery_S^{\texttt{mem}}$ operator, where S is the SQL statement translated from $P^{\texttt{db}}$. As a special case, when $SendQuery_S^{\texttt{mem}}$ is the root operator of e in an $T^{\texttt{db}} \leftarrow e$, a temporary table `T` is created directly from the results of S without any data transfer. Each maximal in-memory sub-plan $Q^{\texttt{mem}}$ that is an input to $P^{\texttt{db}}$ is moved to a new $\texttt{U}^{\texttt{db}} \leftarrow Q^{\texttt{mem}}$, which creates a temporary table `U` in the database and bulk copies the result of evaluating Q into `U`. `U` is subsequently utilized by S. The final plan of the running example as output by the plan-to-SQL translator is shown in Figure 13.

4.8 Alternate Denormalized-sets Plan

To illustrate an alternate rewrite rule for replacing α operators, consider the *denormalized-sets plan* in Figure 14, which is Figure 7 rewritten with the alternate rule, and augmented with source decisions for ease of exposition. The α operator is replaced, but with key differences from the *normalized-sets plan* of Figure 10. As quantified by experiments in Section 5.2, this denormalized-sets plan has two inherent performance limitations as compared to the normalized-sets plan.

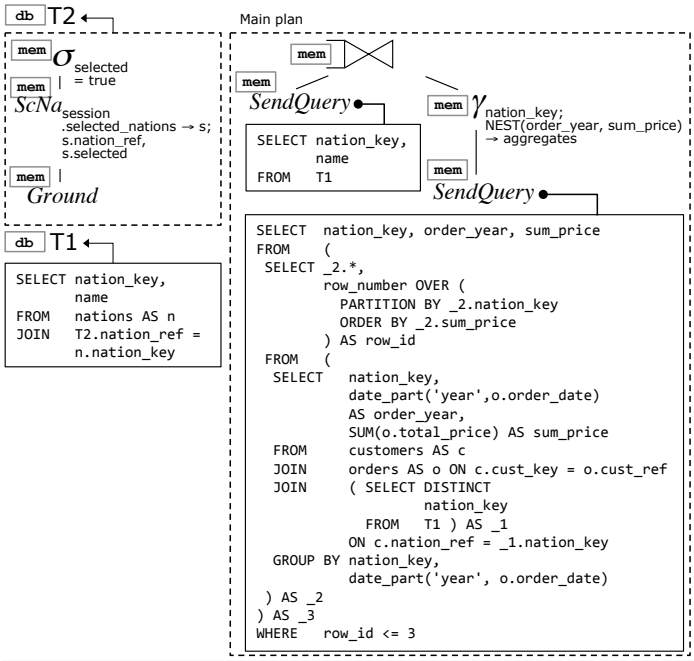


Figure 13: Plan after translating in-database sub-plans to SQL

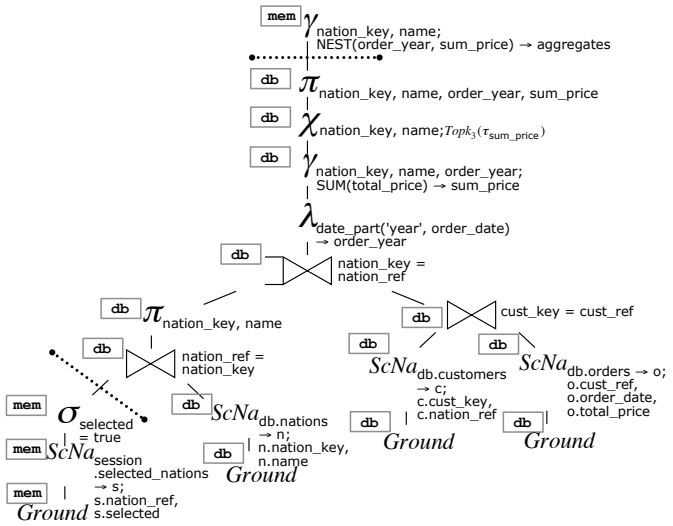


Figure 14: Denormalized-sets plan by alternate rewrite rule

First, observe that for $\alpha_{P \to N}(E)$ where E has correlated attributes $X_1, \ldots, X_n$, there is a functional dependency from $X_1, \ldots, X_n$ to the attributes of N. For example, `nation_key` → `order_year,sum_price`. In the denormalized-sets plan, $\rtimes$ denormalizes intermediate results by joining on $X_1, \ldots, X_n$, thus introducing more non-correlated attributes and/or tuples from its LHS. This creates two types of performance penalties in Figure 14. (1) Horizontal redundancy: $\rtimes$ outputs wider tuples due to the additional `name` attribute, which increases processing costs of the γ_{SUM}, χ, π and γ_{NEST} operators above, as well as the data transfer cost across the cut. (2) Vertical redundancy: $\rtimes$ outputs more tuples than its RHS. If the correlation attributes were other than `nation_key` and have duplicate values in the LHS, each RHS tuple will be copied once per duplicate value, and the additional tuples will similarly increase subsequent processing and data transfer costs. In contrast, the normalized-sets plan avoids these redundancies by adding *Scan* and δ operators (which is reminiscent of the classical semi-join technique in distributed query processing, and deferring $\rtimes$ till after all other operators.

Second, recall that $\rtimes$ is not a commutative operator, and thus restricts join reordering optimizations[6]. In the denormalized-sets plans, $\rtimes$ is decided to be in-database, thus the resulting SQL query specifies (`T3` $\bowtie$ `nations`) $\rtimes$ (`customers` $\bowtie$ `orders`), where `T3` (not shown in figure) is a temporary table bulk loaded with `session.selected_nations` tuples. Due to $\rtimes$, no join reordering occurs, thus the database first executes the expensive join between `customers` and `orders`, which produces as many tuples as `orders`. In contrast, the normalized-sets plans places $\rtimes$ above the memory-specific γ_{NEST}, causing the distributor to decide $\rtimes$ to be in-memory. The resulting SQL query only has $\bowtie$ operators, i.e. a three-way join `T1` $\bowtie$ `customers` $\bowtie$ `orders`, where `T1` has `nation_key`s of selected nations (Figure 13). The database reorders joins and first executes `T1` $\bowtie$ `customers`, which is an order of magnitude more selective than `customers` $\bowtie$ `orders`, thus producing a substantially more efficient plan.

5. EXPERIMENTAL EVALUATION

We demonstrate the benefits of holistic optimization by experimentally evaluating Collage's performance speedup on the running example of Figure 1. All experiments are run with PostgreSQL 9.2 on Mac OS X 10.6, using 512 MB of RAM for PostgreSQL's memory buffers to cache random disk accesses, and 1 GB of RAM for OS X's disk cache to cache sequential disk accesses. With a scale factor of 3 for the TPC-H benchmark [22], the data generator produces a database that contains 25 nations, 450,000 customers and 4,500,000 orders. Running times are based on warm buffers/caches.

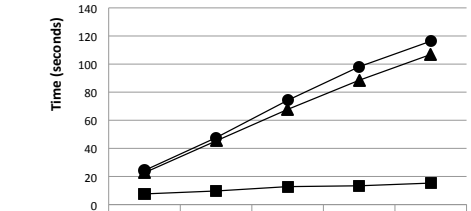
Four queries/plans are evaluated: (1) `RoR` denotes the tuple-at-a-time queries produced by the RoR code of Figure 3 (2) `T` denotes the initial tuple-at-a-time plan in Figure 7, (3) `NS` denotes the normalized-sets plan in Figure 10 as output by the apply-plan rewrite rule (4) `DS` denotes the denormalized-sets plan in Figure 14 as output by the alternate rewrite rule.

Section 5.1 shows the speedup of set-at-a-time over tuple-at-a-time semantics by comparing the running time of the rewritten `NS` against the initial `T` and mainstream `RoR`. Section 5.2 validates our choice of rewrite rules by demonstrating the superiority of `NS` over the alternate `DS`.

5.1 Tuple-at-a-time vs Set-at-a-time plans

In Figure 15, the number of selected nations k is varied from 5 to 25 to show the speedup of set-at-a-time versus tuple-at-a-time semantics at different sizes of results displayed on pages. Due to their tuple-at-a-time semantics, both `RoR` and `T` issue k queries, i.e. one query per selected nation. The join between `customers` and `orders`, which is an expensive operation in each query since the low join selectivity causes Postgresql to scan both tables sequentially, is evaluated k times in total. Not only do the running times

[6]While database researchers have shown that rewriting $\rtimes$ and other join operators to a "generalized-join" operator can enable further join reordering [12], we did not find a RDBMS that implements this optimization.



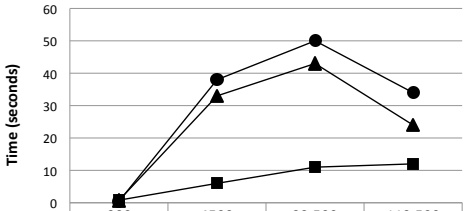
	5	10	15	20	25
RoR	24.3	47.6	74.2	98.1	116.3
T	22.7	45.3	67.7	88.5	106.8
NS	7.6	9.7	12.8	13.4	15.3
Speedup (NS vs RoR)	3.2	4.9	5.8	7.3	7.6

k: Number of selected nations displayed on page

Figure 15: Speedup at different number of selected nations displayed on page

increase proportionally with k, the linear increase also dominates the constant term. That is, the running time for 25 nations is almost 5 times that of 5 nations.

On the other hand, the running time of NS increases with a much gentler slope. Since NS scans tables customers and orders once respectively, it saves more scans over RoR and T with increasing k: speedup of NS over RoR increases from 3.2x to 7.6x as k increases from 5 to 25. Note that NS increases slightly with k as increasing selected nations will also increase the number of intermediate result tuples input by operators, in particular $\gamma_{\texttt{SUM}}$. These data points illustrate that visualizations on pages that require more nested data to be retrieved (such as the bar charts of Figure 1) will benefit increasingly and dramatically from set-at-a-time optimizations.



	900	4500	22,500	112,500
RoR	0.4	38	50	34
T	0.7	33	43	24
NS	0.8	6	11	12
Speedup (NS vs RoR)	0.5	6.3	4.5	2.8

Fanout: Number of customers per nation in database

Figure 16: Speedup at different database join selectivities (due to different number of nations in database)

Figure 16 shows how different join selectivities in the database affects speedup. To vary join selectivities, different TPC-H database instances are created by changing the total number of nations from 25 to 500, 100, 20 and 4 respectively. The total number of customers and orders remain the same, but customers are evenly re-distributed among the available nations. This produces a logarithmic scale of nation-to-customer fanouts: 900, 4,500, 22,500 and 112,500. The number of selected nations is fixed at 10, except in the last fanout where only 4 nations can be displayed. Higher nation-to-customer fanout corresponds to lower selectivity for nation $\bowtie$ customers.

At the high selectivity of fanout 900, PostgreSQL uses index lookups against both customers and orders for all of RoR, T and NS. In this case, NS does not increase performance over RoR, and in fact incurs a 0.1s penalty over T due to the added partition χ operator. However, at the low selectivities of fanouts 4,500 and 22,500, RoR and T incurs sequential scans of customers and index scans of orders, whereas NS incurs sequential scans of both customers and orders. This is a classical example of when low selectivity leads to sequential scans being more efficient than multiple index lookups, thus NS shows a speedup of 4.5-6.3x. At the lowest selectivity of fanout 112,500, all plans incur sequential scans for both tables, thus the reasons for speedup is identical to that of Figure 15. These data points illustrate that when databases increase in size, the consequent increase in fanouts (and hence lowering of selectivities) in aggregation queries will lead to increased benefits from set-at-a-time optimizations. Furthermore, since NS wins by large margins for expensive queries and remains competitive for inexpensive ones, we believe that the apply-plan rewriting should be applied whenever possible.

5.2 Denormalized-sets vs Normalized-sets plans

	Fanout							
	900		4,500		22,500		112,500	
Width (bytes)	DS (s)	NS	DS	NS	DS	NS	DS	NS
4	0.8	0.9	12.4	6.4	15.0	11.1	16.2	11.9
20	0.8	0.9	17.3	6.4	18.0	11.1	18.0	12.4
100	0.9	0.9	17.0	6.4	19.2	11.2	22.2	12.0
500	1.0	0.9	25.8	6.4	21.0	11.2	25.5	12.0
2,500	2.6	0.9	65.7	6.4	41.6	11.1	58.0	12.0

Table 1: Speedup at different nation table widths and database join selectivities

Table 1 compares the running times between DS and NS. Two parameters are varied: (a) the width of a tuple in the nation table along a logarithmic scale, by changing the character length of the name column (b) the fanout of customers per nation, by using the identical methodology and fanout values as in Section 5.1. The number of selected nations is also fixed at 10. Both parameters respectively highlight the two performance limitations of DS as explained in Section 4.8.

From top-to-bottom for increasing tuple widths, the running time of DS increases proportionally to the tuple width, whereas NS is unaffected. This illustrates the horizontal redundancy experienced in DS, and the effectiveness of the *Scan* and δ operators in NS for avoiding these redundancies. Note that vertical redundancy is not demonstrated here, since the correlated attribute nation_key is unique in the LHS of $\rtimes$, otherwise DS will have an even higher penalty.

From left-to-right for increasing fanouts (i.e. lower nation-customer selectivities), the running times for both DS and NS increase. Recall that DS executes (T3 $\bowtie$ nations) $\rtimes$ (customers $\bowtie$ orders) without the benefit of any join reordering. With a lower selectivity, the $\rtimes$ on the large intermediate result customers $\bowtie$ orders becomes more expensive, thus DS increases in running time. On the other hand, NS executes (T1 $\bowtie$ customers) $\bowtie$ orders after join reordering, thus it is affected to a lesser extent by the lower selectivity since customers is an order of magnitude smaller than customers $\bowtie$ orders.

These data points illustrate that when tables become wider and fanouts increase, NS will have increased benefits over DS, thus providing empirical justification for Collage's apply-plan rewriting to output NS plans.

6. RELATED WORK

Database and programming language research work has investigated the problem of efficient and/or single-language access, both before and after the advent of ORMs.

Single access via semistructured query languages Strudel [11] showed that content-publishing web pages can be conveniently specified using a semistructured XML query language. Later, XPERANTO [21] and SilkRoute [10] addressed the optimization of XML queries that produce nested results by accessing an SQL database. The semistructured query is executed by emitting one or more SQL queries to the database and consequently combining the results.

Unlike Collage, they do not consider analytics queries (i.e., queries with group-by, top-k), memory-database distribution or navigations into external objects (and the complications they introduce on the reduction of tuple-at-atime to set-at-a-time). Due to the limited (with respect to Collage) scope of their queries and the respective limited scope of their optimization decisions, they do not need Collage's algebra rewriting-based query processing engine.

In the context of non-distributed select-join nested queries (i.e., nested queries over just an SQL database, group-by and top-k) XPERANTO, Silkroute and Collage still have differences. XPERANTO produces an SQL query q_i for each collection c_i of the nested query (be it the top level connection or a nested collection) by left outer-joining all the FROM-WHERE clauses on the path to c_i. Unlike Collage, each sub-query inefficiently uses the entire table produced for the parent collection although only the correlating attributes are required.

Tuple-at-a-time to set-at-a-time reduction in SQL WHERE

Many research efforts have been devoted to the sub-query decorrelation in the `WHERE` clause since the 1980's. Kim [19] first developed the query transformation algorithms to rewrite nested queries into equivalent, flat queries which can be processed more efficiently. Dayal [9], Muralikrishna [20], Ganski-Wong [14] extended the work to support subqueries with aggregates and more than one level of nesting.

Galindo-Legaria and Joshi [13] represents the subqueries in SQL Server using apply operator (as Collage does), and then removing the correlations by transforming them into joins,semi-joins, outer-joins, etc., according to a set of rules built upon. Other works [1, 6, 5] consider the problem when correlated subqueries occur within positive and negative existential (ANY, EXISTS, IN) or universal (ALL) quantification. The techniques all involve extending the standard relational algebra. We consider them as complementary to work ours.

Optimization of database-accessing application code A second line of research adopts a conventional architecture, where code written in a conventional application programming language (e.g., Java) issues SQL statements directly or indirectly through the use of an ORM. These works analyze the code and, consequently, they generally rewrite the application code to improve data access performance.

Loop lifting [16] is a technique originally developed to handle XQuery FLOWR construct using relational algebra plans. Later, it was generalized and extended to the FERRY compilation framework [18], with a richer data model and more language constructs to handle queries in more general programming languages.

Finally, StatusQuo [7] automatically partitions the application computation between the application server and database server.

7. REFERENCES

[1] M. O. Akinde and M. H. Böhlen. Efficient computation of subqueries in complex olap. In *ICDE*, pages 163–174, 2003.

[2] Anonymized due to double-blind requirements.

[3] Anonymized due to double-blind requirements.

[4] Anonymized due to double-blind requirements.

[5] A. Badia. Computing SQL queries with boolean aggregates. In *Data Warehousing and Knowledge Discovery*, pages 391–400. Springer, 2003.

[6] B. Cao and A. Badia. A nested relational approach to processing SQL subqueries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 191–202. ACM, 2005.

[7] A. Cheung, O. Arden, S. M. A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. *CIDR*, 2013.

[8] W. R. Cook and A. H. Ibrahim. Integrating programming languages and databases: What is the problem. In *In ODBMS.ORG, Expert Article*, 2005.

[9] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[10] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *ACM SIGMOD Record*, volume 30, pages 103–114. ACM, 2001.

[11] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with Strudel. *VLDB J.*, 9(1):38–55, 2000.

[12] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74, 1997.

[13] C. A. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD Conference*, pages 571–581, 2001.

[14] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD Conference*, pages 23–33, 1987.

[15] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*, chapter 5, pages 205–241. Prentice-Hall, 2009.

[16] T. Grust. Purely relational flwors. In *XIME-P*, volume 37, 2005.

[17] T. Grust and M. Mayr. A deep embedding of queries into Ruby. In *ICDE Conference*, 2012.

[18] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *SIGMOD Conference*, pages 1063–1066, 2009.

[19] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[20] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, volume 516, 1989.

[21] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, 10:133–154, 2001.

[22] Tpc-h homepage. http://www.tpc.org/tpch/.

[23] Wikipedia. Template processor, 2013. Accessed Aug 16 2013. http://en.wikipedia.org/w/index.php?title=Template_processor&oldid=567590946.

[24] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, page 32, 2006.