Semi-structured Rewriting

Jules Testard
UC San Diego
jtestard@cs.ucsd.edu

Yannis Papakonstantinou
UC San Diego
yannis@cs.ucsd.edu

ABSTRACT

To be filled in later.

1. INTRODUCTION

There has been a rise in interest over the last few years in querying and producing semi-structured, hierarchical data. Document databases, which query and produce documents with nested structure (such as MongoDB, AsterixDB and CouchDB), are being increasingly utilized for use cases ranging from operational and analytical intelligence to web application development. As a testimony to this growth, the NoSQL database market is expected to increase to over four billion dollars by 2020 [1]. ETL Data integration middlewares that are used to insert into those database also process queries with nested results. The queries that have to be answered by those systems are inherently non-relational, and require query optimization techniques adapted to hierarchical data.

This paper focuses one particular optimization, query decorrelation for semi-structured query processing. Query decorrelation has been studied extensively in the relational context [6, 2, 9, 3] where the subquery occurs in the `WHERE` clause; while fewer works have focused on the `SELECT` clause, given its limited applicability in the context of SQL [7]. Previous solutions have however been provided in the context of XQuery [8], where subqueries in the `return` clause are allowed. Nevertheless, the existing solutions have some performance limitations in the presence of 1) duplicates in the outer query and 2) joins in the subquery. Moreover, no prior published work has tackled the problem when those subqueries are analytical in nature (i.e. they contain grouping/aggregation, sorting and top-k operations). We address both problems in this paper by introducing a set of algebraic equivalences which improve on prior work.

One first difficulty which arises when developing query optimization techniques for semi-structured data is the lack of standard query language across the semi-structured query processors. Ong. et al [] showed that the various languages of SQL-on-Hadoop, NoSQL and NewSQL databases can be accurately modeled by a unifying, SQL backwards-compatible language called SQL++. SQL++ is most easily defined by removing restrictions from SQL semantics. Of particular interest is the ability to have any kind of SQL subquery in the `SELECT` clause, potentially creating nested results. We will be using SQL++ for examples throughout the paper.

```
Promotion(p_promo_sk, p_promo_name,
    p_item_sk ,...);
WebSale(ws_promo_sk, ws_sales_price ,...);

SELECT p1.p_promo_name, (
    SELECT p2.p_promo_name,
           SUM(ws_sales_price) AS price
    FROM Promotion AS p2, WebSales
    WHERE ws_promo_sk = p2.p_promo_sk
    AND @p1.p_item_sk = p2.p_item_sk
    GROUP BY p2.p_item_sk
    ORDER BY price DESC
    LIMIT 3
) AS top_web_sales
FROM Promotion AS p1
LIMIT L;
```

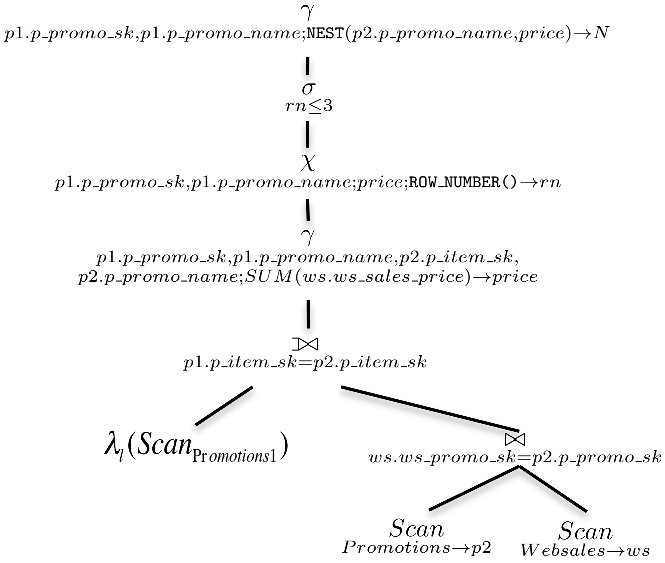
Listing 1: Analytical query and schema for a large web store

On figure 1 is shown a SQL++ nested query typical of an analytical intelligence use case over a large web store. The schema for the query is taken from the BigBench [5] benchmark. For the first L promotions, an analyst wants to identify the top 3 promotions over the same item in terms of web sales revenue. Notice that the 1) the subquery is parameterized with the outer promotion table reference `@p1` and 2) the subquery will return a collection of tuples with two attributes each, one for the promotion name and one for the revenue, which would make it invalid in the SQL context.

The execution that is closest to the query formulation is to compute the `top_web_sales` value using a separate subquery for each promotion tuple which satisfies the `WHERE` clause condition. This is commonly considered an poor strategy, as it involves *tuple-at-a-time* processing of the subquery. A much more efficient query execution strategy is made possible by decorrelating the subquery, allowing the entire set of subqueries to be processed at once, hence the name *set-at-a-time* execution.

On figure 1 is shown an algebraic representation of the denormalized set-at-a-time execution. This execution plan improves on top of the work of May et al. [8] by providing set-at-a-time semantics for SQL++ analytical clauses `GROUP BY`, `ORDER BY` and `LIMIT` within the subquery. The portion of the plan below the left outer join or reminiscent of classic query decorrelation. The outer query's `FROM` and `LIMIT` clauses are processed on the left hand side of the left outer join, using the top-k (λ_k) operator. On the right hand side of the left outer join, the `FROM` clause of the subquery is processed. The left outer join then processes the subquery processes the subquery's `WHERE` clause predicate for all correlated attributes at once. The top portion of the algebraic plan allows the processing of the analytical clauses by : 1) the subquery's `GROUP BY` clause is processed using a γ operator which groups on the primary key of $p1$, 2) `ORDER BY` and `LIMIT` clauses are processed using the windowing operator χ which computes row numbers for each partition, followed by a selection which filters rows whose row number is greater than the value of the `LIMIT` clause of the subquery. Finally, the creation of the `top_web_sales` collection is handled by the $\gamma_{\texttt{NEST()}}$ operator.

Figure 1: Denormalized Set-At-A-Time execution

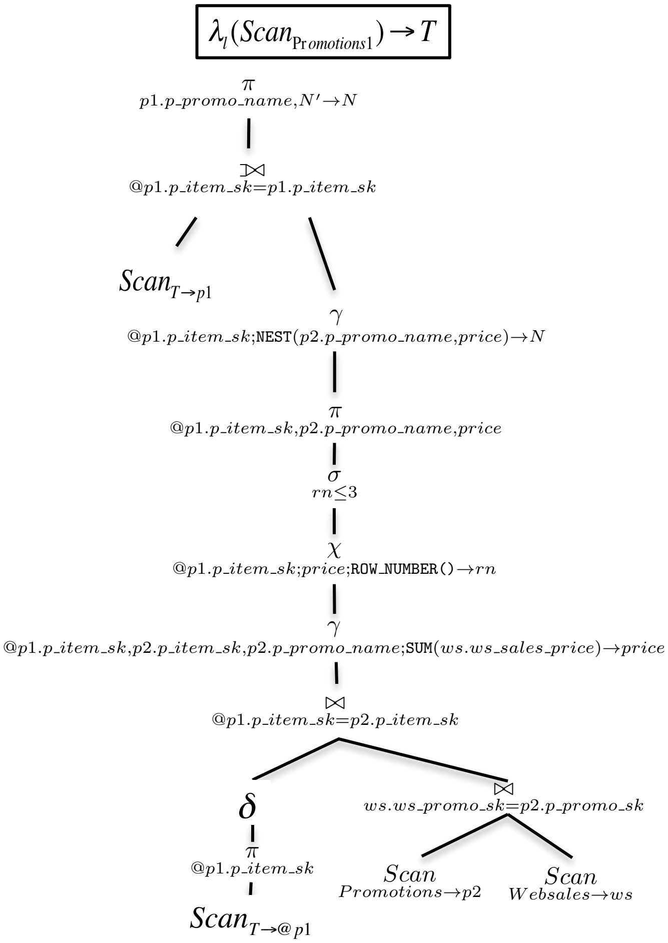

γ
$p1.p\_promo\_sk,p1.p\_promo\_name;\texttt{NEST}(p2.p\_promo\_name,price)\rightarrow N$
σ
$rn\leq 3$
χ
$p1.p\_promo\_sk,p1.p\_promo\_name;price;\texttt{ROW\_NUMBER()}\rightarrow rn$
γ
$p1.p\_promo\_sk,p1.p\_promo\_name,p2.p\_item\_sk,$
$p2.p\_promo\_name;SUM(ws.ws\_sales\_price)\rightarrow price$
⟕
$p1.p\_item\_sk=p2.p\_item\_sk$
$\lambda_l(Scan_{\text{Promotions1}})$
$\bowtie$
$ws.ws\_promo\_sk=p2.p\_promo\_sk$
$Scan$
$Promotions\rightarrow p2$
$Scan$
$Websales\rightarrow ws$

Notice that in the running example, there is a functional dependency from the correlated attribute `p_item_sk` to the attributes `p_promo_name` and `price` of `top_web_sales`. The ⟕ denormalizes intermediate results by joining on `p_item_sk`, thus introducing more non-correlated attributes from its left hand side. This creates two types of performance penalties: 1) horizontal redundancy: the ⟕ operator outputs unnecessarily wide tuples due to the additional $p1.p\_promo\_name$ attribute (which isn't used until the $\gamma_{\texttt{NEST}}$ operator), which increases the cost of the $\gamma_{\texttt{SUM}}$ and χ operators; 2) vertical redundancy: the ⟕ operator outputs redundant tuples if the correlated attribute $p1.p\_item\_sk$ is not unique, which also unnecessarily increases operator costs. Moreover, it introduces a requirement that the tuples from the output of left hand side of the left outer join be unique, since a key is required to process the γ and χ operators. In the running example, the key is $p1.p\_promo\_sk$.

Jules: Show NSAAT and how it improves upon DSAAT by outlying performance improvements.

We introduce a normalized-set rewriting on figure **??**. It introduces the $Scan$ and δ operators (which is reminiscent of the classical semi-join technique in distributed query processing), and deferring the ⟕ until all analytical operators are processed.

Figure 2: Normalized Set-At-A-Time execution


$\lambda_l(Scan_{\text{Promotions1}})\rightarrow T$
π
$p1.p\_promo\_name,N'\rightarrow N$
⟕
$@p1.p\_item\_sk=p1.p\_item\_sk$
$Scan_{T\rightarrow p1}$
γ
$@p1.p\_item\_sk;\texttt{NEST}(p2.p\_promo\_name,price)\rightarrow N$
π
$@p1.p\_item\_sk,p2.p\_promo\_name,price$
σ
$rn\leq 3$
χ
$@p1.p\_item\_sk;price;\texttt{ROW\_NUMBER()}\rightarrow rn$
γ
$@p1.p\_item\_sk,p2.p\_item\_sk,p2.p\_promo\_name;\texttt{SUM}(ws.ws\_sales\_price)\rightarrow price$
$\bowtie$
$@p1.p\_item\_sk=p2.p\_item\_sk$
δ
π
$@p1.p\_item\_sk$
$Scan_{T\rightarrow @p1}$
$\bowtie$
$ws.ws\_promo\_sk=p2.p\_promo\_sk$
$Scan$
$Promotions\rightarrow p2$
$Scan$
$Websales\rightarrow ws$

Jules: Outline and contributions

Jules: The key requirement for DSAAT is only necessary because the $\gamma_{\texttt{NEXT()}}$ is happening above the left outer join. If the $\gamma_{\texttt{NEXT()}}$ is placed below the left outer join, the key requirement is no longer necessary but the performance may degrade because aggregation happens before the left outer join, which may be filtering

Jules: @Yannis: One question that could arise is whether it is necessary that the input data to the query be relational, since it is in our running example. In particular, what happens if the correlation attributes are nested or missing for some or all tuples.

Jules: The May paper [8] uses the χ symbol for their apply plan operator and the α symbol to mean the head of a list. I hope our use of the same symbols for different operators won't confuse our readers early on.

Jules: Unfortunately, there is no equivalent to the `LIMIT` keyword in the SQL standard, each DBMS having it's own syntax. My introduction assumes the semantics of `LIMIT` are obvious.

2. DATA MODEL AND ALGEBRA

In this paper, we use the SQL++ semi-structured data model and query language [10]. We only retain here the most relevant features for our setting. The complete version of the language is available in [10].

2.1 The SQL++ data model

SQL++ queries input and output collections of values. Formally, a *SQL++ value* is either

1. a *primitive value*, such as the string `'abc'` or the integer `7`.
2. 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 a SQL++ value.
3. a *collection* $[v_1, \ldots, v_n]$, where each v_i is recursively a SQL++ value. A collection may be ordered (then called an *array*) or unordered (then called a *bag*).
4. the `null` value.

2.2 The SQL++ Query Language

The SQL++ language is backwards compatible with SQL. As such, we define its semantics by *removing* semantic restrictions from SQL:

- Unlike SQL's `FROM` clause variables, which bind to tuples only, the `FROM` clause variables of SQL++ may bind to any arbitrary SQL++ value.

- SQL++ is fully composable in the sense that subqueries can appear anywhere, potentially creating nested results when they appear in the `SELECT` clause.

- Unlike SQL's aggregate functions which output scalar values, SQL++'s aggregate function may output any value. Of particular interest to the optimizations presented in this paper is the `NEST`(e) function, whose output is a collection. Formally, just like any aggregate function, it is evaluated on each group created by `GROUP BY` clause. For each group, it returns a collection in which each element corresponds to the evaluation of expression e for one tuple of the group.

- Since SQL++ tuples may have attributes whose value is itself a tuple with attribute of its own, SQL++ allows path navigation of arbitrary depth. Formally, a *tuple path navigation* $t.a$ from a tuple t with existing attribute a returns the value of a in t. If a does not exists in t, then $t.a$ returns the `null` value.

  Jules: I don't believe `MISSING` is required for this rewriting.

Tuple-at-a-time and Set-at-a-time nesting Notice that a SQL++ query that produces nested collections can be formulated in two ways: either using a nested query in the `SELECT` clause, or using the `NEST()` aggregate function. In the first case, the nested query evaluating to a nested collection is evaluated for each tuple of the main query individually, and therefore we call this query formulation tuple-at-a-time nesting. In the second case, the nested collections are created all at once by the `GROUP BY` clause, and therefore we call this query formulation set-at-a-time nesting.

2.3 The query pattern

We consider here SQL++ queries which exhibit the following pattern :

```
1   SELECT ... , (
2       SELECT $p_1$ AS $u_1, \ldots, p_q$ AS $u_q$ ,
3           $a_1(\ldots)$ AS $A_1$ , ... , $a_m(\ldots)$ AS $A_m$ ,
4           $w_1(\ldots)$ OVER (PARTITION BY $v_1^1, \ldots, v_{n_1}^1$
                ORDER BY $oc_1$) AS $W_1$ ,
5           ... ,
6           $w_l(\ldots)$ OVER (PARTITION BY $v_1^l, \ldots, v_{n_l}^l$
                ORDER BY $oc_l$) AS $W_l$ ,
7           ($P_1$) AS $N_1$ , ... , ($P_k$) AS $N_k$
8       FROM $\hat{F}(c_1, \ldots, c_n)$
9       WHERE $w$
10      GROUP BY $g_1, \ldots, g_i$
11      HAVING $h$
12      ORDER BY $o_1, \ldots, o_j$
13      LIMIT $l$
14    ) AS N
15  FROM ...
16   ...
```

Listing 2: Query Pattern

On listing 2, we denote lines 1, 15 and 16 to be the *outer query*, and lines 2-14 to be the *inner query*. The `SELECT` clause items other than the inner query and the SQL clauses below and including the outer query's `FROM` clause may contain any arbitrary SQL++ expression, thus are only shown as ... on the code fragment above. The inner query has the following characteristics:

1. the variables produced by the outer query which may be in any SQL++ expression in the inner query. We will denote those variables as $c_1, \ldots, c_n$.

2. **Jules:** @Yannis: The original draft for this paper also included in the input pattern to the rewriting a list of zero of more assignments $CS_1 \leftarrow E_1, \ldots, CS_p \leftarrow E_p$ in which $E_1, \ldots, E_p$ are plans, possibly correlated with variables from the outer plan. I left them out here. We could add them through a `WITH` clause in the query pattern.

3. The inner query may contain any of the following SQL++ clauses : `WHERE`, `GROUP BY`, `PARTITION BY`, `HAVING`, `ORDER BY`, `LIMIT`. Moreover, the items in any of those clauses may be any arbitrary SQL++ expression.

   Jules: @Yannis: the `OVER` and `PARTITION BY` clauses on figure 2 are not mentioned in the SQL++ query language paper [10]. I don't know if we can just put them there without further introduction.

4. In the `SELECT` clause, expressions $a_1, \ldots, a_m, w_1, \ldots, w_l$ may be arbitrary SQL++ aggregation functions bound to variables $A_1, \ldots, A_m, W_1, \ldots, W_l$ respectively. Moreover, $P_1, \ldots, P_k$ may be any nested queries bound to variables $N_1, \ldots, N_k$ (These won't be rewritten away by the rule, and need subsequent calls to the rewriting rule to be removed from the plan.). Finally, $p_1, \ldots, p_q$ may be arbitrary expressions bound to variables $u_1, \ldots, u_q$.

5. The `FROM` clause expression $\hat{F}(c_1, \ldots, c_n)$ may either be: (i) a single from item (name of a stored collection, collection literal, variable/path navigation mapped to

a collection or a subquery) or (ii) any kind of join (`LEFT|RIGHT|FULL INNER|OUTER|CROSS JOIN`) (for conciseness we consider the cross product a special kind of join) between two items supported in the `FROM` clause, recursively. In both cases, if some from items are themselves subqueries or joins, these subqueries can be correlated through the attributes $c_1, \ldots, c_n$.

2.4 The SQL++ Algebra

We present here an algebra for SQL++. This is a purely logical algebra, and physical execution of this algebra is discussed further in section 5. An algebraic 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 use the previously computed temporaries $T_1, \ldots, T_{i-1}$. The result of P is the SQL++ collection resulting from the evaluation of the *result expression* e, which may use any temporary $T_1, \ldots, T_n$.

The algebraic operators involved in the SQL++ algebra expressions input and output a collection (called binding collection) of tuples (called binding tuples). This collection may be ordered (then called an array) or unordered (then called a bag). Each binding tuple maps accessible variables (i.e. variables in the scope) to their corresponding value. Each value may itself by any SQL++ value (including a tuple or a collection).

The majority of SQL++ algebra operators are extensions of operators well-known from conventional SQL processing (cf. textbooks [4]). While conventional operators input and output collection of tuples with primitive or scalar attribute values, SQL++ extends algebraic operators to allow the binding attribute values to also be tuples and collections.

The list of standard operators comprises CartesianProduct (or CrossProduct) $\times$, Union $\cup$, Intersection $\cap$, Difference $-$, Selection σ_c, InnerJoin $\bowtie_c$, FullOuterJoin $\mathbin{⟗}_c$, LeftOuterJoin $\mathbin{⟕}_c$, SemiJoin $\ltimes_c$, AntiSemiJoin $\overline{\ltimes}_c$, Projection $\pi_{p_1 \to u_1, \ldots, p_q \to u_q}$, Sort $\tau_{o_1, \ldots, o_j}$ (where $o_1, \ldots, o_j$ is the list of ordering terms, which initially appears in the SQL++ `ORDER BY` clause), limit λ_l (which outputs the first l binding tuples of its input), duplicate elimination δ, group-by $\gamma_{g_1, \ldots, g_i; a_1(.) \to A_1, \ldots, a_m(.) \to A_m}$ (where $g_1, \ldots, g_i$ is the list of grouping terms, which initially appears in the `GROUP BY` clause and $a_1, \ldots, a_m$ are the aggregate functions, which initially appear in the `HAVING` and `SELECT` clauses).

In the remainder, given a binding tuple t, 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 has all the attribute name/value pairs of t as well as the attribute name a mapped to the value v.

The **Ground** operator Ground is the only accepted leaf of a plan. It always outputs a single empty binding tuple $\{\}$.

The **Scan** operator $\text{Scan}_{S \mapsto s}$ is used to iterate over collections. For each input binding tuple t, it iterates over the collection C bound to S and for each element $v \in S$, it outputs the binding tuple $t\#\{s : v\}$. Notice that if the child of Scan is a Ground, it really behaves like an iterator over the collection S (in such cases we do not write Ground explicitly, for conciseness). However, if it is any other operator (e.g. another Scan), it unnests the collection C with respect to each input binding tuple.

The **ApplyPlan** operator $\alpha_{P \mapsto N}$ is the algebraic counterpart of tuple-at-a-time nesting. For each input binding tuple t, the ApplyPlan operator evaluates the plan P to a value v and outputs the tuple $t\#\{N : v\}$. In general, P is a correlated plan $P(t)$, such that variables of the input binding tuples appear as *parameters* in P wherever constants or variables can appear in uncorrelated plans. In this case, each v is the result of evaluating $P \backslash t$, i.e. the plan P in which all the parameters have been replaced by their corresponding value for the current binding tuple t. Notice that if P corresponds to a standard `SELECT` query, each v is a collection and the ApplyPlan effectively nests a collection in every binding. If a SQL++ clause C requires the evaluation of a nested query SQ (e.g. `SELECT (`SQ`) AS s ...`), the algebraic translation $\text{alg}(C)$ of this clause is composed of the corresponding SQL++ operator on top of an ApplyPlan (e.g. $\pi_{\_v \mapsto s}(\alpha_{\text{alg}(SQ) \mapsto \_v}(\ldots))$). Notice that this corresponds to an eager evaluation of nested queries, since all the nested queries used by an operator are evaluated before the operator itself. This algebraic translation does not produce correct results when the nested queries are used by short-circuiting functions (e.g. `CASE WHEN`, `AND`) and possibly produce runtime errors (e.g. division by zero), but in our experience such use cases are very uncommon. The treatment of such nested queries are out of the scope of this paper.

The **GroupBy** operator $\gamma_{g_1, \ldots, g_i; a_1(.) \mapsto A_1, \ldots, a_m(.) \mapsto A_m}$ behaves like SQL's grouping operator: it partitions the input binding tuples according to the evaluation $v_{g_1}, \ldots, v_{g_i}$ of the grouping expressions $g_1, \ldots, g_i$, computes the evaluation $v_{A_1}, \ldots, v_{A_m}$ of the aggregate functions $a_1, \ldots, a_m$ on each partition and outputs, for each partition, the binding tuple $\{g_1 : v_{g_1}, \ldots, g_i : v_{g_i}, A_1 : v_{A_1}, \ldots, A_m : v_{A_m}\}$. SQL++ extends SQL by allowing aggregate function that return complex values, such as the `NEST(`e`)` aggregate function. This particular function returns, for a given partition of input binding tuples, a collection of tuples which includes the evaluation $e(t)$ for each input binding tuple t in that partition. Using `NEST()` as aggregate function in the GroupBy operator is the algebraic counterpart of set-at-a-time nesting.

Like in SQL, total aggregation queries, that is, queries with aggregations in the `SELECT` clause but no `GROUP BY` clause (e.g. `SELECT count(*) FROM customers`) correspond to a special version $\gamma^T_{a_1(.) \mapsto A_1, \ldots, a_m(.) \mapsto A_m}$ of the GroupBy operator. $\gamma^T_{a_1(.) \mapsto A_1, \ldots, a_m(.) \mapsto A_m}$ behaves like $\gamma_{a_1(.) \mapsto A_1, \ldots, a_m(.) \mapsto A_m}$ (a GroupBy operator without grouping attributes, that creates a single partition), with the following exception: if γ inputs an empty binding bag, it outputs a empty binding bag, whereas if γ^T inputs an empty binding bag, it outputs a binding bag with a single binding tuple containing default values $\text{default}(f_i)$ for each aggregate function $f_1, \ldots, f_n$. For instance, default(`SUM()`) is `null` and default(`COUNT()`) is `0`.

The **PartitionBy** operator $\chi_{x_1, \ldots, x_{n1}; oc; w(.) \mapsto W}$ enables support of the SQL 2003 `PARTITION` feature and window functions. In addition, χ is important for the optimizations discussed in this paper, as one of them introduces a χ operator in the rewritten plan (cf. Section 3.2). Formally, χ (i) partitions the input binding tuples according to the evaluation $v_{x_1}, \ldots, v_{x_{n1}}$ of the grouping expressions $x_1, \ldots, x_{n1}$ then (ii) sorts the binding tuples within each partition according to the optional `ORDER BY` clause oc (identical to a standard SQL `ORDER BY` clause, but only sorts the bindings within each partition) then (iii) computes the evaluation v_W of the window function w for each binding tuple of each partition and finally (iv) outputs, for each binding tuple t of each partition, the binding tuple $t\#\{W : v_W\}$. For example:

$\chi_{\texttt{nation};\tau_{\texttt{sales},\texttt{DESC}};row\_number()\mapsto\texttt{rank}}$

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

$=$

nation	name	sales	rank
USA	Joe	6	1
China	Zhao	8	1
China	Fu	4	2

3. ALGEBRAIC FORMULATIONS

3.1 Tuple At A Time

On figure 3 is shown an algebraic plan corresponding to a direct translation of the query pattern into SQL++ algebra, which we denote as the *tuple-at-a-time* (TAAT) algebraic formulation. This plan is the plan which semi-structured databases would generate initially, before any algebraic rewriting optimizations are performed. For this reason, we also call this formulation the initial plan. The plan on the left side of the figure corresponds to the translation of the outer query (denoted *outer plan*), while the plan (denoted as P) in the dashed-lined box is the translation of the inner query (denoted as *inner plan*).

Outer Plan The expression E of the outer plan corresponds to the algebraic translation of the outer query SQL clauses below and including the `FROM` clauses, while the expression G is the translation of the `SELECT` clause items other than the inner query. The apply plan operator α is correlated by variables $c_1, \ldots, c_n$ while $z_1, \ldots, z_{q'}$ are those variables produced by E which are required by operators in G.

Inner Plan Within the inner plan (denoted as P), the value of $F(c_1, \ldots, c_n)$ corresponds to the translation of the `FROM` clause expression $\hat{F}(c_1, \ldots, c_n)$. Next, operators σ_w, $\gamma_{g_1,\ldots,g_i;a_1(.)\mapsto A_1,\ldots,a_m(.)\mapsto A_m}$, σ_h, $\tau_{o_1,\ldots,o_j}$, λ_l correspond to the algebraic translation of the query pattern's `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY` and `LIMIT` clauses, respectively. Finally, operators $\chi_{x^1_1,\ldots,x^1_{n1};oc^1;w^1(.)\mapsto W^1}, \ldots, \chi_{x^l_1,\ldots,x^l_{nl};oc^l;w^l(.)\mapsto W^l}$, $\alpha_{P_1\mapsto N_1}, \ldots, \alpha_{P_k\mapsto N_k}$ correspond to the `PARTITION BY` statements and subqueries present in the `SELECT` clause.

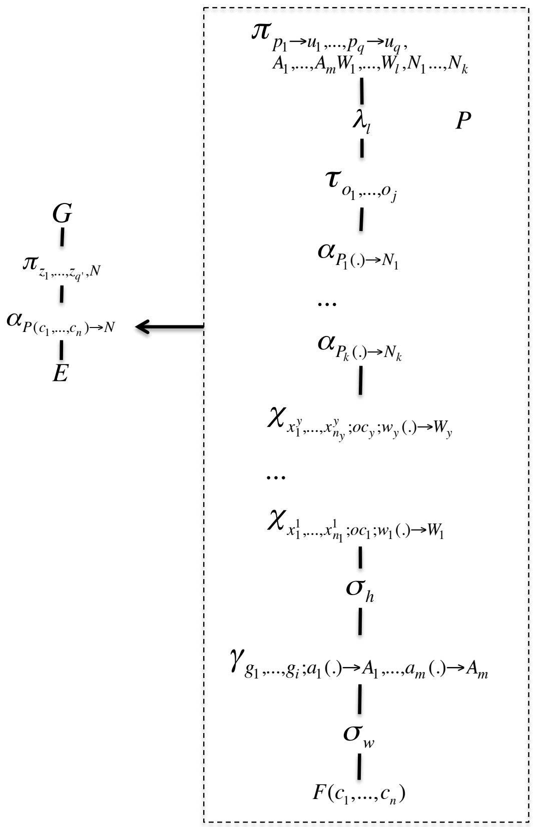
Jules: @Yannis: one tricky thing to notice here is the position of the Apply plan (α) operators, which reflect here the possible presence of nested queries in the input's `SELECT` clause. However, given we allow any arbitrary SQL++ expression in the `WHERE`, `GROUP BY`, etc clauses, we could, for example, have a subquery in the `WHERE` clause. Those would not be accounted for in this translation of the plan.

The tuple-at-a-time formulation shown above constitutes only one way to formulate the query pattern. In particular, it evaluates the inner plan P once for every binding tuple in the output of E. We present next two alternative algebraic formulations which evaluate the inner query for every binding tuple at the same time, which we denote as a set-at-a-time execution. We distinguish the *normalized* and *denormalized* set-at-a-time executions.

3.2 Normalized Set-at-a-time

The normalized set-at-a-time (NSAAT) formulation of the query pattern is shown on the figures 4 and 5. We obtain this formulation by rewriting the tuple-at-a-time formulation. Instead of evaluating a very similar (only differing by the value of the correlated attributes) inner query for each tuple in the output of E, it will (i) introduce a cross product between E and the leaves of the inner plan $P(c_1, \ldots, c_n)$ (ii) execute the inner plan only once on the output of this cross product (iii) group the tuples in the output of this inner plan according to the tuple of E they correspond to (iv) create the nested collections using the `NEST()` aggregate function (v) join back with the original E to preserve cardinality. Thus, the normalized set-at-a-time formulation replaces the tuple-at-a-time nesting operator α with a set-at-a-time nesting operator $\gamma_{\texttt{NEST}}$. The details of the rewriting are as follows:

Figure 3: Tuple-At-A-Time Formulation (TAAT)


$\pi_{p_1\to u_1,\ldots,p_q\to u_q, A_1,\ldots,A_m W_1,\ldots,W_l, N_1\ldots,N_k}$
λ_l
P
$\tau_{o_1,\ldots,o_j}$
G
$\pi_{z_1,\ldots,z_{q'},N}$
$\alpha_{P_1(.)\to N_1}$
...
$\alpha_{P(c_1,\ldots,c_n)\to N}$
E
$\alpha_{P_k(.)\to N_k}$
$\chi_{x^y_1,\ldots,x^y_{n_y};oc_y;w_y(.)\to W_y}$
...
$\chi_{x^1_1,\ldots,x^1_{n_1};oc_1;w_1(.)\to W_1}$
σ_h
$\gamma_{g_1,\ldots,g_i;a_1(.)\to A_1,\ldots,a_m(.)\to A_m}$
σ_w
$F(c_1,\ldots,c_n)$

Outer Plan The rewriting of the outer plan is shown on figure 4. The output of E is assigned to a temporary table T. The plan P is rewritten to $r(P, T)$ (as explained below). The result of this rewritten plan goes to a $\gamma_{\texttt{NEST(.)}}$ operator that groups according to the correlated attributes $c_1, \ldots, c_n$ and nests all the binding tuples with the same correlated values. As we will see next, the group of binding tuples nested with a given set of correlated values $v_{c_1}, \ldots, v_{c_n}$ corresponds exactly to those bindings which would be outputted by the original P if parameterized with values $v_{c_1}, \ldots, v_{c_n}$. Next, a $⟖$ operator joins this back with the original T and a final π operator ensures that the extra attributes created by the outer join are projected out. π also replaces the `null` values introduced by the outer join with empty collections (in the figure, N' stands for the translation of the SQL++ statement `CASE WHEN N IS NULL THEN [] ELSE N`). Note that if the original inner plan is a total aggregation query, the $\gamma^T_{a_1(.)\to A_1,\ldots,a_m(.)\to A_m}$ is rewritten into $\gamma^T_{c_1,\ldots,c_n;a_1(.)\to A_1,\ldots,a_m(.)\to A_m}$. In this case, the top π operator replaces `null` N bindings with a collection containing a single tuple with default values for each total aggregate function in the original query.

Figure 4: Normalized Set-At-At-Time Formulation (NSAAT)

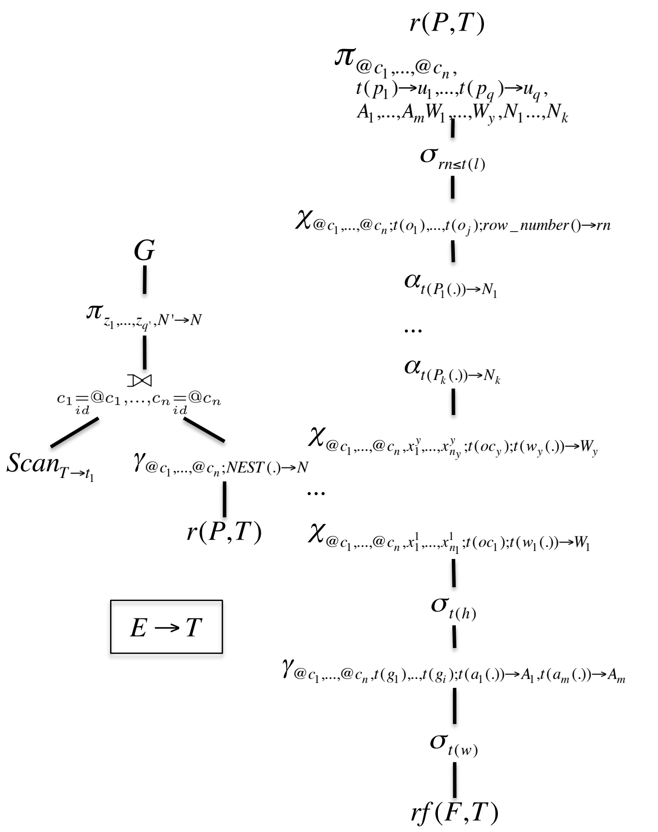
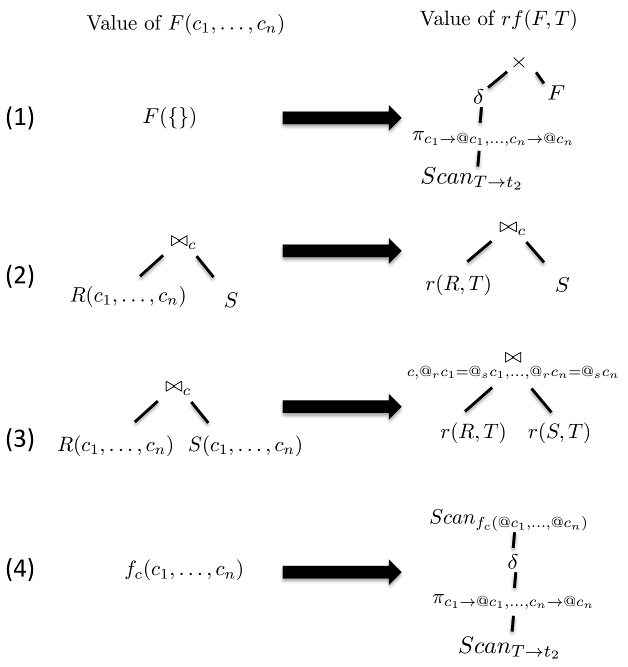

$r(P,T)$
$\pi_{@c_1,...,@c_n, t(p_1)\to u_1,...,t(p_q)\to u_q, A_1,...,A_m W_1,...,W_y, N_1...,N_k}$
$\sigma_{rn \leq t(l)}$
$\chi_{@c_1,...,@c_n; t(o_1),...,t(o_j); row\_number()\to rn}$
$\alpha_{t(P_1(.))\to N_1}$
...
$\alpha_{t(P_k(.))\to N_k}$
$\chi_{@c_1,...,@c_n, x_1^y,...,x_{n_y}^y; t(oc_y); t(w_y(.))\to W_y}$
...
$\chi_{@c_1,...,@c_n, x_1^1,...,x_{n_1}^1; t(oc_1); t(w_1(.))\to W_1}$
$\sigma_{t(h)}$
$\gamma_{@c_1,...,@c_n, t(g_1),...,t(g_i); t(a_1(.))\to A_1, t(a_m(.))\to A_m}$
$\sigma_{t(w)}$
$rf(F,T)$
G
$\pi_{z_1,...,z_{q'}, N'\to N}$
$\bowtie_{c_1 \underset{id}{=} @c_1,...,c_n \underset{id}{=} @c_n}$
$Scan_{T\to t_1}$
$\gamma_{@c_1,...,@c_n; NEST(.)\to N}$
$r(P,T)$
$E \to T$

Figure 5: FROM clause rewriting



Jules: The approach above could be problematic because it does not distinguish between the **nulls** created by the outer join from the **nulls** pre-existing on the right hand side. This is not a problem here because all binding tuples from the right hand side will have a non-null **N**

We next describe how to rewrite the inner plan's FROM clause (figure 5). The rewriting $r(P,T)$ of the nested plan P introduces a cross product between the distinct correlated attributes of T an the leaves of P. This is done by rewriting $rf(F,T)$ of the FROM clause F of P. The rewriting of F requires a case analysis (for conciseness, we treat $A \times B$ in the initial plan as $A \bowtie_{true} B$ and do not explain it further): i) If F is not correlated, it is rewritten into a simple cross product between F and $\delta(\pi_{c_1,\ldots,c_n}(T))$. A π operator keeps only the correlated attributes $c_1,\ldots,c_n$ renamed into $@c_1,\ldots,@c_n$ (case 1 of figure 5). ii) If F is one correlated subquery $R(c_1,\ldots,c_n)$, it is recursively rewritten into $r(R,T)$ (not shown in figure). iii) if F is a join between two FROM items, but only one is correlated $R(c_1,\ldots,c_n)$, then only R is recursively rewritten into $r(R,T)$ (if it is a subquery) or $rf(R,T)$ (if it is a join). iv) If F is a join between two correlated branches $R(c_1,\ldots,c_n)$ and $S(c_1,\ldots,c_n)$, both are recursively rewritten to $r(R,T)$ / $rf(R,T)$ and $r(S,T)$ / $rf(S,T)$ if they are subqueries/joins respectively. Moreover, the join condition c is modified to enforce that the bindings outputted by the join correspond to the same values of correlated attributes on both sides (case 3 of figure 5, @ symbols have additional subscripts to distinguish the renamings on the left and right hand side). v) If F is a correlated expression (e.g. a path starting from a correlated variable and evaluating to a collection), it is rewritten into an unnesting Scan of $t(f_c(c_1,\ldots,c_n))$ on top of $\delta(\pi_{c_1,\ldots,c_n}(T))$. vi) Finally, if a join condition in F is correlated, the rewriting corresponds to the case when both branches are correlated and the join condition is rewritten into $t(c)$ (not shown in the figure). Notice that the value of $rf(F,T)$ in cases (1) and (4) from figure 5. The duplicate elimination is only required if there may be duplicate correlate attributes in the input. In the case where the set of correlated attributes includes a key to the output of expression E ($key(E) \subseteq \{c_1,\ldots,c_n\}$), the δ operator can be eliminated.

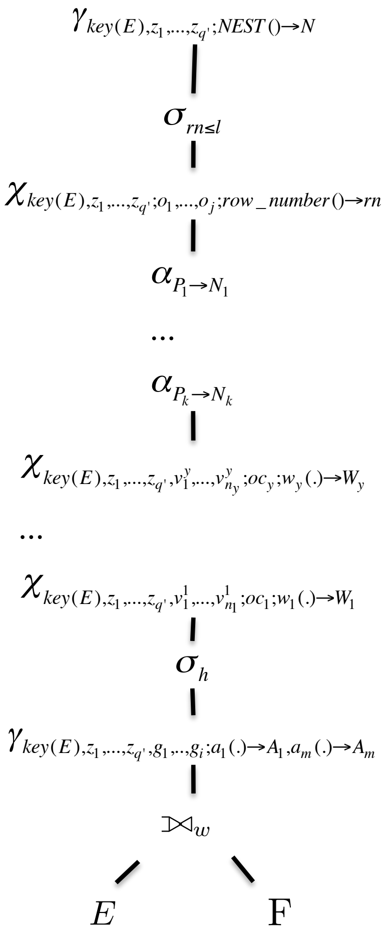
Inner Plan The rewriting of the inner plan P is shown on figure **??**. If X is or an expression of plan P correlated by attributes $c_1,\ldots,c_n$, we denote $t(X)$ the same expression X in which the attributes $c_1,\ldots,c_n$ are substituted with variables $@c_1,\ldots,@c_n$. i) The top π operator of P is modified to also output the attributes $@c_1,\ldots,@c_n$ necessary for the $\gamma_{\mathtt{NEST}(.)}$ and $⟗$ operators above. iii) The nested plans $P_1,\ldots,P_k$ of eventual α operators in P are rewritten to $t(P_1),\ldots,t(P_k)$. iv) The eventual τ and λ operators are replaced with two other operators: on the one hand, a χ operator that groups according to $c_1,\ldots,c_n$, orders according to the $t(o_1),\ldots,t(o_j)$, then computes the row number of each tuple in its partition. The row number can then be used with a σ operator to simulate the λ (note that, depending on the implementation, if the γ operator at the top of the rewritten plan breaks the ordering between the binding tuples coming from the χ, the NEST function will have to sort again the tuples in each nested collection according to the row number). v) The grouping attributes of the eventual χ operators of P are modified to also group by $@c_1,\ldots,@c_n$. vi) The HAVING clause condition h is rewritten to $t(h)$. vii) The grouping attributes of the eventual γ operator of P are modified to also group by $@c_1,\ldots,@c_n$. viii) The WHERE condition w is rewritten to $t(w)$.

3.3 Denormalized Set-at-a-time

The denormalized-set-at-a-time (DSAAT) formulation of the query pattern is shown on figure **??**. We also describe this pattern rewriting of the TAAT formulation.

E has to be a set. It can't just be a

Figure 6: Denormalized Set-At-At-Time Formulation (DSAAT)


$\gamma_{key(E),z_1,...,z_{q'};NEST()\rightarrow N}$
$\sigma_{rn\leq l}$
$\chi_{key(E),z_1,...,z_{q'};o_1,...,o_j;row\_number()\rightarrow rn}$
$\alpha_{P_1\rightarrow N_1}$
...
$\alpha_{P_k\rightarrow N_k}$
$\chi_{key(E),z_1,...,z_{q'},v_1^y,...,v_{n_y}^y;oc_y;w_y(.)\rightarrow W_y}$
...
$\chi_{key(E),z_1,...,z_{q'},v_1^1,...,v_{n_1}^1;oc_1;w_1(.)\rightarrow W_1}$
σ_h
$\gamma_{key(E),z_1,...,z_{q'},g_1,...,g_i;a_1(.)\rightarrow A_1,a_m(.)\rightarrow A_m}$
$\bowtie_w$
E
F

4. COST MODEL

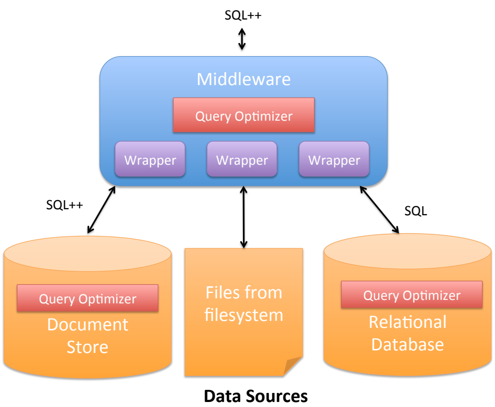
We show that the proposed rewriting leads to better performance in a large variety of "real-world" data analytics scenarios. In this section, we present an abstract analysis of the cost of the TAAT, NSAAT and DSAAT algebraic formulations on execution environments typically utilized for those scenarios. For this purpose, we introduce a theoretical cost model focused on network overhead and disk access based on a number of (listed) assumptions regarding network, storage, indices and relational operators which are common across those execution environments.

4.1 Setup

4.1.1 Execution Environment

We argue the query rewriting presented in this paper has value on a variety of different query processors for semi-structured data. We consider two broad categories of query processors: document stores and data integration middlewares. We evaluate the performance improvement of our rewriting using a theoretical architecture (shown on figure 7) which is representative of both categories. This architecture has two components: a data processing *middleware*, and a set of zero or more *data sources* with various levels of *capabilities*.

Figure 7: Query Processor Architecture



Middleware: the data processing middleware answers SQL++ queries by executing algebraic plans over the data sources. It is capable of processing any SQL++ algebraic operator. To process SQL++ algebraic operators, the middleware has access to a number of physical operators (algorithms) which are listed on table 1. The middleware also has access to a query optimizer, which allows it to choose which physical implementation to use for processing of a given algebraic operator. It is also capable of delegating a portion or all of the plan to a data source, if : 1) the data source is capable of processing all operators in that plan and 2) all stored collections accessed in that plan are located at that data source. Once a plan is delegated to a datasource, the wrapper for that datasource converts the algebraic plan into a query expressed in that datasource's native query language.

Data Sources: they contain the data queried by the middleware. We consider here only three kinds of data sources: a file system, a traditional relational database system and document stores. Each data source has its own set of characteristics which we describe next:

- **File System**: Any file system which is accessible by the middleware process. This source is not capable of processing any SQL++ operator.
- **RDBMS**: Any relational database system which has access to the physical operators shown on table 1. For simplicity, we do not assume the RDBMS has access to any other physical operators. An RDBMS is only capable of processing *purely relational* operators. By purely relational, we mean operators whose output are regular rows of atomic scalars (i.e. they do not contain any nesting or heterogeneity).
- **Document Store**: we consider any document store which is capable of processing any SQL++ algrebraic

plan. We assume the document store also has access to the physical operators shown on table 1. In the case where the document store is the single datasource, we consider the middleware optional (the user may query using the document store directly).

We assume that all query processors considered to be using single machine setups. We do not consider the cases where operators can run as parallel tasks of a cluster in a share-nothing architecture. We also assume all databases (including the middleware) have the ability to spill to disk in case the size of an intermediate results exceeds the available memory. In practice, document stores as well as integration middlewares employ complex optimizations that cannot be comprehensively accounted for in an analytical model. As such, the computation costs and associated speedups reported in this section should be used as rough estimates of the actual cost.

The middleware answers queries from a single source at a time (i.e. we do not consider complex cases where the data is joined from multiple source in the middleware). This assumption is of particular importance when

Jules: @Yannis: I see a problem here when we reach experiments. I only know two kinds of middlewares (FORWARD middleware and Informatica), none of which can spill to disk without invoking a "supporting" database.

4.1.2 Notations

- A plan context $P[]$ denotes a plan with one "hole"; i.e. a plan in which an n-ary operator has $n-1$ operators and a special character $[]$ as children, and $P[Q]$ represents the plan P in which the hole has been replaced with the subplan Q. For example, for $P[] = \pi_p([])$ and $Q = \sigma_w(R)$, we have $P[Q] = \pi_p(\sigma_w(R))$.

- $cost(P)$ denotes the cost of executing a plan P and $cost(P[], Q)$ denotes the cost of executing the plan $P[Q]$ assuming Q has already been evaluated and its result is accessible is accessible at no cost.

- $network(P)$ denotes the cost of sending the result of P to a target database system through a network with throughput K_n. Notice that we only consider "internal" network cost. For example, in the Hadoop ETL setting, we do not consider the cost of sending the end result to the target database.

- $|P|$ denotes the number (resp. average number) of binding tuples outputted by a subplan (resp. correlated subplan) P. Conversely $\#_f(P)$ is the number (resp. average number) of memory frames required to store the result of subplan (resp. correlated subplan) P. Thus we have $\#_f(E) = \lceil \frac{|E|.T_E}{f_s} \rceil$, where T_E is the tuple width and f_s is the frame size. In the remaining of the paper, we approximate this expression to $\#_f(E) = \frac{|E|}{f_s}$.

  Jules: This approximation could cause a problem, since tuples containing nested collections may have arbitrary size and ignoring this can lead to incorrect size estimates. Likewise, one of the advantages of NSAAT over DSAAT is the decrease in tuple width, which this assumption would eliminate.

- $attr(E)$ denotes the set of attributes in the output of expression E.

- $join\_size(E, F, w)$ denotes the size of the output (in memory frames) of a join with condition w. We have $(E, F, e.c = f.c \cap w')) = \#_f \frac{|E||F|}{max(V(c,E),V(c,F))} s_{w'}$, where $c \in attr(E) \cap attr(F)$ and $s_{w'}$ is the selectivity of the condition w'. Notice that we do not distinguish between the size of the output of an inner and the outer join.

- Aggregation functions are said to be $bounded - state$ if the size of their output is bounded. Traditional aggregation functions such as `COUNT(.)` and `AVERAGE(.)` belong to this category, while `NEST(.)` does not, since the size of its output grows with the size of its input.

- We denote M_{DB} as the memory budget for each operator in database system DB (e.g. $M_{PostgreSQL}$ when considering the PostgreSQL system). This memory budget is expressed in terms of number of memory frames. We assume all operators in a given database system are entitled to the same memory budget. In cases where there is only one database system for an execution environment, we simply say M.

- We denote $V(c, E)$ as the number of distinct values of attribute c in the output of expression E.

- We call the *reach* of a query the amount of data which *must* be read from database instance in order to accurately answer a query, while in the presence of all possible indices.

  Jules: I would be surprised if I am the first to ever introduce this term.

- We denote $merge(R)$ as the cost of sorting using the merge-sort join technique beyond the first pass. Note that $merge(R) = \#_f(R)$ when $R < M^2$ (only two passes).

4.1.3 Assumptions

To make the analysis simpler, we first make the following assumptions (and discuss them later on):

- We assume the size of a memory frame $\#_f(F)$ equals the size of a disk block. This way we have $SeqScan(R) = \lceil \frac{R.T_R}{B} \rceil = \lceil \frac{R.T_R}{f_s} \rceil = \#_f(R)$.

- We only look at the number of blocks, not at the time required to fetch those blocks (not distinguishing between random and sequential IO).

- We assume all indices are available and will be used by the query optimizer of the database system if it chooses to do so.

- We assume no skew in correlation attribute values. This way, we assume that the value of attribute N of a apply plan operator $\alpha_{P(c_1,\ldots,c_n) \to N}$ will have the same size, no matter which values correlated attributes $c_1, \ldots, c_n$ take.

- We assume $\alpha_{P(c_1,\ldots,c_n) \to N}$ are applied sequentially, with the fully memory budget available to *each* execution of the plan P.

Table 1: Operator Costs

Physical Operator / Color Code	Read Cost	Transform Cost	Write Cost	Restrictions / Characteristics
$T \leftarrow R$ (Copy)	$\#_f(R)$	0	$\#_f(R)$	
$\sigma_w, \pi_p, \lambda_l$	0	0	0	pipelined
$Scan^{\textbf{sequential}}(R)$	$\#_f(R)$	0	0	
$Scan^{\textbf{index}}_{c_r=X}(R)$	$\frac{\lvert R\rvert}{V(c,R)}$	0	$\#_f(\frac{\lvert R\rvert}{V(c,R)})$	R is a table, $c_r \in attr(R)$, index on $R.c$ and X is a scalar
$Join^{\textbf{memory\_hash}}_{c_r=c_s}(R, S)$	$\#_f(S) + \#_f(R)$	0	$join\_size(R, S, w)$	$\#_f(R) < M_{DB}$, R already loaded in memory
$Join^{\textbf{right\_index}}_{c_r=c_s}(R, S)$	$\#_f(R)$	$\lvert R\rvert\frac{\lvert S\rvert}{V(c,S)}$	$join\_size(R, S, w)$	$\#_f(R) < M_{DB}$, R already loaded in memory
$Join^{\textbf{sort\_merge}}_{c_r=c_s}(R, S)$	$\#_f(R) + \#_f(S)$	$\#_f(R) + \#_f(S) + merge(R) + merge(S)$	$join\_size(R, S, w)$	
$Join^{\textbf{sort\_merge}}_{c_r=c_s}(R, S)$	$merge(R) + merge(S)$	0	$join\_size(R, S, w)$	inputs R and S are sorted on join attributes
$\tau(R), \chi(R), \gamma^{\textbf{SB}}_{NEST}(R)$	$\#_f(R)$	$\#_f(R) + merge(R)$	$\#_f(R)$	
$\gamma^{\textbf{PC}}_{NEST}(R)$	$\#_f(R)$	0	$\#_f(R)$	input R is sorted on g
$\gamma^{\textbf{PC}}_{BS}(R), \delta_{PC}(R)$	$\#_f(R)$	0	$\#_f(\delta(\pi_g(R))$	input R is sorted on g
$\gamma^{\textbf{SB}}_{BS}(R), \delta_{SB}(R)$	$\#_f(R)$	$\#_f(\delta(\pi_g(R)) + merge(R)$	$\#_f(\delta(\pi_g(R))$	
$\alpha_{P\rightarrow N}(R)$	$\#_f(R)$	$\lvert R\rvert cost(P)$	$\#_f(R)$	Tuple size increased, number of tuples reduced

- In all scenarios, we assume the size of the result of the inner query (of the query pattern, that is) is expected to be small, given that nesting big collections isn't practical and does not represent a real use case.

- The `FROM` clause F of the inner plan P is a join of n stored collections. We assume only one of these collections, denoted with R, is correlated with the outer plan through a selection $\sigma_{c_e=c_r}(R)$, where c_e and c_r attributes of E and R, respectively. We denote F' the plan context such that $F = F'[\sigma_{c_e=c_r}(R)]$. Finally the expression $rf(F, T)$ is such that the join order of F has not changed. That is, $rf(F, T) = F'[T \bowtie_{c_e=c_r} R]$.

- We assume the presence of both `GROUP BY` and `ORDER BY` clause in the input nested query.

  Jules: @Yannis: I believe it is not necessary to explicitly describe queries with either only `GROUP BY` or only `ORDER BY` in the nested query separately.

- We make the assumption that $\frac{|F|}{Mf_s} \leqslant f_s$, which is true anytime $|F|$ is smaller than f_s times the memory size. Given f_s typically ranges in the kB to MB range, this assumption holds unless $|F|$ is bigger than one million times the memory size, which we don't consider realistic in practice.

- We make the assumption that no stored collection or intermediate result ever exceed M^2, which reflects realistic use cases.

- In the case of an index being used by a $Scan^{index}$ or $Join^{right\_index}$ operator, we assume this index is stored entirely in memory, and does not take space in the memory budget allocated for the query.

Note that the running example query follows these assumptions.

4.1.4 Operators considered

Our cost model provides a taxonomy of physical operators implementations which can be used to form execution plans (those are shown on table 1). Each row on table 1 corresponds to a implementation of a algebraic operator. The cost associated with using this operator is split into three sections: read cost, transform cost and write cost. Write cost is only paid if output is too large to fit in memory and is not the final result. Read cost is paid if input is not already in memory. In case of joins, if one of the inputs is already in memory but not both, then only the cost of reading the input not in memory is paid. Transform cost is paid if size of the input is greater than memory. The cost of the execution of a given operator is the sum of all three costs. Finally, the restrictions column indicates under which conditions can a particular implementation be used. Therefore, we define the *cost* of executing an algebraic operator to be the cheapest cost among competing implementations of that operators which meet the restrictions.

4.1.5 Cost Breakdowns

We next describe precisely the cost breakdown of expressions TAAT, NSAAT and DSAAT.

$$\begin{aligned} cost(TAAT) =& |E|(cost(Scan(R)) + cost(F'([]), Scan(R)) + \\ & + cost(\tau(\gamma_{inner}([])), F)) \end{aligned}$$

The TAAT plan evaluates $|E|$ times the inner plan P, which itself is the addition of the following costs: i) the cost of evaluating the `FROM` clause (which requires scanning collection $\sigma_{e_c=r_c}(R)$ and joining with other collection in plan context F') and ii) the cost of evaluating the inner `GROUP BY` (denoted γ_{inner}) and `ORDER BY` clauses on the output of the `FROM` clause.

$$\begin{aligned} cost(NSAAT) =& cost(E \rightarrow T) + cost(rf(F,T)) \\ & + cost(\gamma_{\texttt{NEST}}(\chi(\gamma_{inner}([])), rf(F,T)) \\ & + cost(⟕([T,]), r(P,T)) \end{aligned}$$

The cost of evaluating the NSAAT plan is the addition of the following costs : i) the cost of the assignment $E \rightarrow T$ ii) the cost of evaluating $rf(F,T)$ iii) the cost of the γ_{inner} aggregation, the χ partition-based ordering and the $\gamma_{\texttt{NEST}}$ nesting aggregation iv) the cost of the final outer join between T and $r(P,T)$. Note that evaluating $rf(F,T)$ requires performing an inner join between the distinct correlated values fetched from T and the expression F resulting in a the following cost : $cost(rf(F,T)) = cost(\delta(T)) + cost(\bowtie([, R]), \delta(T)) + cost(F'([]), \bowtie(\dots))$.

$$\begin{aligned} cost(DSAAT) =& cost(F) + cost(⟕([E,]), F) \\ & + cost(\gamma_{\texttt{NEST}}\chi(\gamma_{inner}([])), ⟕(\dots)) \end{aligned}$$

Finally, the cost of evaluating the DSAAT plan is the addition of the following costs: i) the cost of expression F, ii) the cost of the initial outer join between E and F iii) the cost of the γ_{inner} aggregation, the χ partition-based ordering as well as the $\gamma_{\texttt{NEST}}$ aggregation.

Note that these costs assume that selection pushdown and cross-product removal optimizations have already occurred. Given the cost of computing expression E is paid no matter what, it is not included in the above cost expressions. Note as well that pipelined operators (λ, σ and π) are not included either. Finally, some systems may introduce additional costs to those expressions, such as network costs in middleware settings.

4.2 Scenarios

We next evaluate the performance of the TAAT, NSAAT and DSAAT algebraic formulations in three different scenarios. For each scenario, we describe the characteristics of the query pattern and then discuss and compare the cost of the expressions for different ranges of $V(c_e, E)$ and $V(c_r, F)$, the number of distinct correlated attribute values for expressions E and F, respectively. Finally, we summarize the cheapest implementations for each scenario in the appendix cost tables 2, 3 and 4.

4.2.1 Scenario 1: Large Reach, Small Output

This situation is typical of analytical intelligence use cases, in which human analysts query large datasets and read the results of their queries to obtain valuable insight. The running example scenario belongs to this category.

- The result of such use cases are typically small (thus the ”small output”), since they have to be human-readable. As a result, the size of E is expected to be tiny in database terms (may be less than 100). In this situation, the cost $E \rightarrow T$, $\delta(T)$ and $\gamma_{\texttt{NEST}}$ are insignificant.

- The expression F could be very large (thus the ”large reach”), and exceed the memory budget M many times.

4.2.2 Scenario 2: Large Reach, Large Output

This situation is typical of ETL use cases, in which large datasets are transformed and loaded into semi-structured databases.

- The aim of such use cases is too transform large quantities of data in a single query. The result of such use cases may be well beyond a human readable size. As a result, the size of E is expected to be very large. In this situation, the cost $E \rightarrow T$, $\delta(T)$ and $\gamma_{\texttt{NEST}}$ become significant.

- The expression F could be very large (thus the ”large reach”), and exceed the memory budget M many times.

4.3 Document Store

The physical plans produced by the query optimizer for each formulation are shown on figure X.

4.3.1 Scenario 1

4.3.2 Scenario 2

4.4 Middleware

4.4.1 Scenario 1

4.4.2 Scenario 2

Jules: Index join is only if possible if n is 1

5. EXPERIMENTAL EVALUATION

To be filled in later.

6. USE CASES

The rewriting we present is useful in a wide variety of uses cases, with very different execution environments and query characteristics. We distinguish at least four use cases: 1) an ETL framework used for operational intelligence 2) A big data analytics system used by business intelligence users, 3) a web application which provides analytics visuals for a browser and 4) a cloud web service/API for external applications.

6.1 Operational Intelligence

Operational Intelligence applications collect machine generated data from logging systems and/or application output for the purposes of monitoring and real-time analytics. It involves two distinct tasks: collecting data, then analyzing it. It is often the case that the database systems used for collection and analysis are distinct, and data must be moved from the former to the latter through the use of ETL tools.

It is also often the case that the system used for analysis is a document store such as MongoDB , which stores data as documents with nested structures. One example is click-stream analysis for an online website. To facilitate further analysis, an ETL middleware pre-aggregates daily, hourly and minute hit counts for every web page from a source log file into a target document database using query 4 from appendix B.

6.2 Big Data Analytics System (BDAS)

Business intelligence analysts need to make queries for historical and analytical intelligence. Business intelligence query results are meant to be readable, therefore are small (a few hundred of rows at most) and typically look at the top-k elements which match a particular criteria. Very fast latencies are not typically a requirement for these kinds of queries, but they sift through very large amounts of data. The number of such queries issued is also expected to be small, given that the number of analysts studying the same dataset at the same time is typically limited.

The execution environment utilized in this use case have typically been parallel data warehouses. However, such systems are inadequate when the result of the formulated queries contain nested collections. For example, consider the following query performed on top of a large retail store dataset: Find the top 30 products that are mostly viewed together with a given an input list of products in an online store [1] (query 5 from appendix B). For each product in the input list, a collection of 30 products must be found. A new breed of system used when this happens in such cases semi-structured (and parallel) databases, such as AsterixDB.

6.3 Analytics Visualization

This use case applies to queries issued by analytics/reporting web applications which display nested, aggregated data. Query results for this use case are small enough to be displayed on a computer screen. Moreover, applications tend to have short latency requirements (less than a second) and a large number of users, each requesting multiple distinct visualization renderings as they interact with various UI widgets. As such, visualization queries tend to be numerous and are usually based on small datasets or pre-aggregated data.

For example, consider an analytics web application for a large retail company, which provides its regional and store managers with visuals to help them assess the performance of the clerks working in their store. The application has checkboxes to select clerks working in store X, and produces details about the top K largest orders, in terms of total sales price (query 6 from appendix B). For each clerk in the selected list, a collection of K orders must be retrieved.

6.4 Online Web Services & APIs

This use cases applies to online web services, typically hosted in the cloud, which provides information to be consumed by external services, such as mobile applications or other web services. Three commercial examples of such services would include:

- Algolia: a full text search web service, which stores and indexes customer text data (such as product reviews), and provides an API to query it.
- Keen.IO: a web service which embeds a visual analytics client on the customer's website, and collects/stores the customer data to be visualized.
- Loggly: a log management web service, which stores and indexes customer logs and allows customers to easily query and visualize them.

These web services share a number of characteristics: their queries have typically a low latency requirements, the number of queries answered is large (given each customer can issue an arbitrary number of queries), their APIs only allow a narrow set of highly specialized query patterns and their query results are structured with nesting for program consumption (typically using the JSON data format). Consider a case in which the Loggly web service is used by some large retail website customer. In order to help the customer sift through its page view log, Loggly would provide the per-minute and per-hour page view hit-counts for any given page upon request (query 3 from appendix B). The result would produce one tuple per hour, while each minute hit count would be stored in a nested collection field in the corresponding hour tuple.

7. RELATED WORK

To be filled in later.

8. REFERENCES

[1] M. Asay. Nosql databases eat into the database market, 2015.

[2] 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.

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

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[5] A. Ghazal, T. Rabl, M. Hu, and F. Raab. Bigbench: Towards an industry standard benchmark for big data analytics. In *SIGMOD*, 2013.

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

[7] N. May, S. Helmer, and G. Moerkotte. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, 2001.

[8] N. May, S. Helmer, and G. Moerkotte. Three cases for query decorrelation in xquery. In *Database and XML Technologies*, 2003.

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

[10] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The sql++ query language: Configurable, unifying and semi-structured. Technical report, UCSD, 2014.

APPENDIX

A. COST TABLES

Jules: In table 2 and 3, external sorts also include spilling and reading back from disk in between the two sorts.

[1] corresponds to the query 2 from the TPCx-BB benchmark

Table 2: TAAT plan cost table

Costs	Scenario 1	Scenario 2	Scenario 3
0A: $\sigma(R)$	$\lvert E\rvert Scan^{index}$ if $f_s \leqslant V(c_r, F)$ $\lvert E\rvert Scan^{Sequential}$ else		
0B: $F'[], \sigma(R)$	$\lvert E\rvert(n-1 \text{ joins})$		
0C: $\tau(\gamma_{inner}([])), F$	0 if $\frac{\lvert F\rvert}{V(c_r, F)} \leqslant Mf_s$ $\lvert E\rvert$(2 external sorts) else		

Table 3: NSAAT plan cost table

Costs	Scenario 1	Scenario 2	Scenario 3
2A: $E \rightarrow T$	insignificant		
2B: $\delta(\pi_{c_e}(T))$	insignificant		
2C: $\bowtie([, R]), \delta(\pi_{c_e}(T))$	$Join_{c_e=c_r}^{right\_index}(\delta(\pi_{c_e}(T)), R)$ if $f_s \leqslant V(c_r, F)$ $Join_{c_e=c_r}^{memory\_hash}(\delta(\pi_{c_e}(T)), R)$ else		
2D: $F'([]), \bowtie(\dots)$	n-1 joins		
2E: $\chi(\gamma_{inner}([]), rf(F, T)$	0 if $\frac{V(c_e, E)\lvert F\rvert}{V(c_r, F)} \leqslant Mf_s$ 2 external sorts else		
2F: $\gamma_{\mathtt{NEST}}([]), \chi(\dots)$	insignificant		
2G: $⟕([T,])$	insignificant		

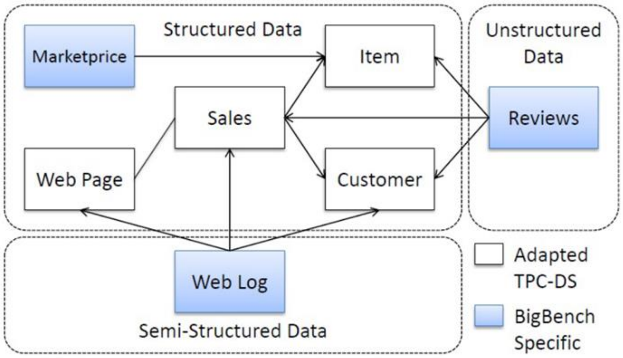
Table 4: DSAAT plan cost table

Costs	Scenario 1	Scenario 2	Scenario 3
4A: F	$n-1$ joins		
4B: $⟕([E,]), F$	$Join_{c_e=c_r}^{right\_index}(E, F)$ if $f_s \leqslant V(c_r, F)$ and F is a relation $Join_{c_e=c_r}^{memory\_hash}(E, F)$ else		
4C: $\chi(\gamma_{inner}([]), ⟕(\dots)$	0 if $\frac{\lvert E\rvert\lvert F\rvert}{V(c_r, F)} \leqslant Mf_s$ 2 external sorts else		
4D: $\gamma_{\mathtt{NEST}}([]), \chi(\dots)$	insignificant		

B. EXAMPLES

Here are provided the queries for the use cases given in section 2. Note that all queries are expressed using the SQL++ language developed at UCSD [10], although those queries would be typically written using an application-specific query language.

Figure 10: TPCx-BB Schema



Example 3 produces pre-aggregated hit-counts per minute and per hour for a webpage Y on a day X. Example 4 produces this output for all distinct {day,webpage} pairs. Examples 3 and 4 are representative of scenarios $S3_S$ and $S3_L$, respectively.

```
Q1(X,Y):
SELECT DISTINCT cs_out.hour,
        SUM(minutes.count) AS count, (
    SELECT cs_in.minute, COUNT(cs_in.key))
    FROM clickstream cs_in
    WHERE cs_out.hour = cs_in.hour
    GROUP BY cs.minute
    ) AS minutes
FROM clickstream cs_out
WHERE day=X AND webpage=Y;
```

Listing 3: Operational Intelligence example, local

```
SELECT DISTINCT cs.day, cs.webpage, SUM(
    hours.count) AS count
    Q1(X=cs.day, Y=cs.webpage) AS hours
FROM clickstream cs;
```

Listing 4: Operational Intelligence example, global

Example 5 corresponds to the query 2 from the TPCx-BB benchmark. It finds the top 30 products that are mostly viewed together with a given list of products in an online store. Note that the order of products viewed does not matter, and ”viewed together” relates to a web_clickstreams `click_session` of a known user with a session timeout of 60min.

```
SELECT p1.name, (
    SELECT p2.name, COUNT(s2.key)
    FROM clickstream cs1,
         clickstream cs2,
         products p2
    WHERE p1.key = cs1.produc
    AND time-delta(cs1.time,cs2.time) < 60
    AND cs2.product = p2.key
    GROUP BY p2.key
    ORDER BY count
    LIMIT 30
) AS top30
FROM selected_products p1;
```

Listing 5: TPCx-BB Query 2

Example 6 finds, for each clerk from a (small) set of selected clerks working in store X, the total price of the top K orders made by that clerk.

```
SELECT c_name, (
    SELECT o_totalprice
    FROM orders
    WHERE c_clerk_key = o_clerk_key
    ORDER BY price DESC
    LIMIT K
) AS top_orders
FROM clerks
WHERE c_clerk_key IN (/*selected from
    application*/)
AND c_clerk_store = X;
```

Listing 6: Top orders from the same clerk for selected orders

Figure 8: Physical Query Plans

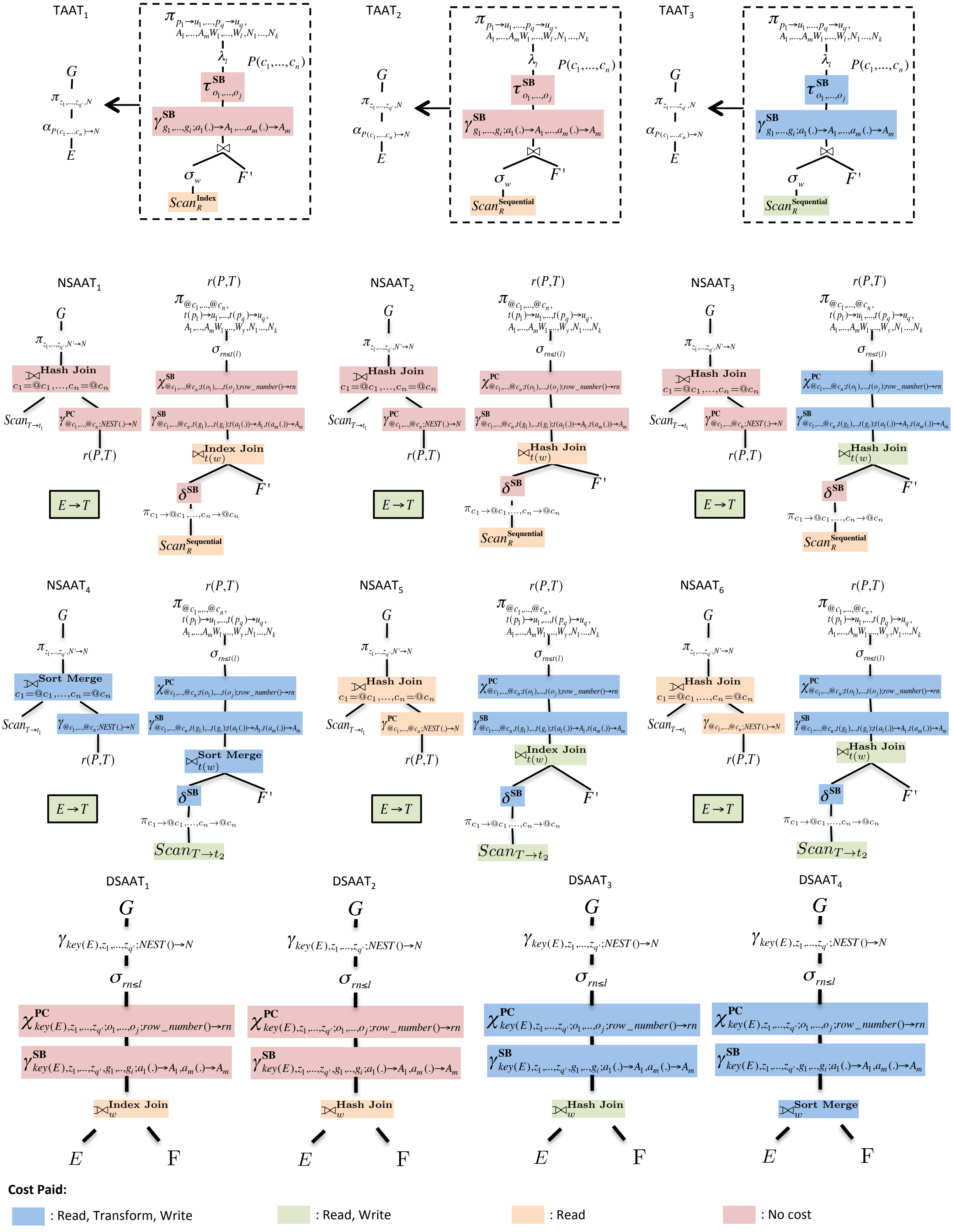


Figure 9: Scenario 1

$V(c_r, F)$

	0	$\frac{\lvert F\rvert}{Mf_s}$	$\frac{V(c_e,E)\lvert F\rvert}{Mf_s}$	$\frac{\lvert E\rvert\lvert F\rvert}{Mf_s}$	f_s	$+\infty$
$TAAT$	$\lvert E\rvert(\frac{\lvert R\rvert}{f_s} + n - 1$ joins $+ 2\frac{F}{V(c,F)f_s} + 5\frac{V(g,0B)}{f_s})$	$\lvert E\rvert(\frac{\lvert R\rvert}{f_s} + n - 1$ joins)			$\lvert E\rvert(\frac{\lvert R\rvert}{V(c,R)} + n - 1$ joins)	
$NSAAT$	$\frac{\lvert R\rvert}{f_s} + \frac{V(c_e,E)\lvert R\rvert}{V(c_r,R)f_s} + n - 1$ joins $+ 2\frac{V(c_e,E)\lvert F\rvert}{V(c_r,F)} + 5\frac{V(c,E)V(g,2D)}{f_s}$		$\frac{\lvert R\rvert}{f_s} + n - 1$ joins		$\frac{V(c_e,E)\lvert R\rvert}{V(c_e,R)} + n - 1$ joins	
$DSAAT$	$\frac{\lvert F\rvert}{f_s} + \frac{\lvert E\rvert\lvert F\rvert}{V(c_r,F)f_s} + n - 1$ joins $+ \frac{2\lvert E\rvert\lvert F\rvert}{V(c_r,F)f_s} + \frac{5\lvert E\rvert V(g,4B)}{f_s}$			$\frac{\lvert F\rvert}{f_s} + n - 1$ joins	$\frac{\lvert E\rvert\lvert F\rvert}{V(c_r,F)} + n - 1$ joins	