Revelation on Demand

Nicolas Anciaux[1] · Mehdi Benzine[1,2] · Luc Bouganim[1] · Philippe Pucheral[1,2] · Dennis Shasha[3]

Abstract Private data sometimes must be made public. A corporation may keep its customer sales data secret, but reveals totals by sector for marketing reasons. A hospital keeps individual patient data secret, but might reveal outcome information about the treatment of particular illnesses over time to support epidemiological studies. In these and many other situations, aggregate data or partial data is revealed, but other data remains private. Moreover, the aggregate data may depend not only on private data but on public data as well, e.g. commodity prices, general health statistics. Our GhostDB platform allows queries that combine private and public data, produce aggregates to data warehouses for OLAP purposes, and reveal exactly what is desired, neither more nor less. We call this functionality "revelation on demand".

Keywords: Confidentiality and privacy, Secure device, Data warehousing, Indexing model, Query processing, Aggregate computation.

1 Introduction

A principal function of data warehouses is to summarize detailed data into various aggregates. Companies frequently provide summaries of confidential data about their activity to trading partners, shareholders, labor unions and even to the public in marketing materials. Government agencies, businesses, and nonprofit organizations also collect, analyze, and report aggregated data over many individuals. Thus, data warehouses frequently map private detailed data to public summary data.

An entire subfield of security addresses the issue of determining whether published summary data reveals detailed data. This problem started attracting attention from the statistical database community more than twenty years ago [1]. More recently, datasets anonymization techniques have been suggested to make information identifying an individual indistinguishable from k-1 other individuals [32]. The work on l-diversity [24] goes one step further by enforcing enough diversity between the privacy sensitive attributes within a same group of tuples. To guarantee that detailed data cannot be recovered from aggregated ones is also a central concern of privacy-preserving data mining techniques [3]. For our purposes, however, we will assume that organization's procedures or laws determine the aggregates that may be revealed. That is, we concentrate on the mechanism not the policy.

Let us consider the following two scenarios. Bob is a traveling salesman and is entrusted with sensitive corporate information about customers and technical specifications. Bob may want to issue aggregate queries that combine public, say Bob's company's product catalog, and sensitive private information about product availability and specifications. He is willing to reveal the aggregate response (according to his company's policy) to the customer data warehouse, but he wants to be sure the sensitive detailed data is not leaked to the untrusted

[1] INRIA Rocquencourt - Le Chesnay, France - <Fname.Lname>@inria.fr
[2] PRiSM Laboratory - University of Versailles, France - <Fname.Lname>@prism.uvsq.fr
[3] Courant Institute of Mathematical Sciences - New York University, USA - shasha@cs.nyu.edu

customer's computing environment. Mary is a physician involved in AIDS research. She manages sensitive data about her patients. A world scale data warehouse has been set up by the research community studying the same disease. Periodically, Mary pushes aggregated data mixing the private data she is managing with external public sources (hospital databases, social databases) to this data warehouse. Similarly to Bob, she needs the assurance that sensitive detailed data (e.g., related to unpublished research results) is not leaked to unscrupulous colleagues. If traditional computing infrastructures could be trusted, Bob and Mary would probably rely on existing database servers to host their data and produce the requested aggregations. Unfortunately, traditional security procedures do not offer the ability to trust privacy policies [11]. Delegating Personal data to a central server increases the risk of disclosure resulting from negligence, piracy, or abusive scrutinization or usage, as defined in [5]. Indeed, any data is exposed to accidental disclosures resulting from negligence. To cite a few, the personal details of 25 million UK citizens have been recently lost inadvertently [36], a PC was recently sold on Ebay with bank customer data including account details and customers' signatures [7], and some of the data published by AOL about Web search queries of 657,000 Americans have been deanonymized [20]. Regarding piracy, even the most defended servers (including those of Pentagon [33], FBI [35] and NASA [12]) are successfully attacked. Finally, personal information is often weakly protected by obscure and loose privacy policies which are presumed accepted when exercising a given service. This fosters ill-intentioned scrutinization and abusive usages justified by business interests, governmental pressures and inquisitiveness among people. Companies like Intelius or ChoicePoint make scrutinization their business.

Recent research does promise additional guarantees under specific assumptions regarding where the trust resides in the system. Hippocratic databases ensure that personal data are used in compliance with the purpose for which the donor gave his consent [2] but require the database server to be trusted. Encrypted databases require either trusting the server [20], [27] or the clients [13], [18] depending on the place decryption occurs. Databases can be entirely hosted in secure hardware [9], [30], [38] but this solution applies only to very small single-user databases.

We propose a very different approach, called GhostDB, to protect sensitive data and produce accurate aggregates. The basic idea is to remove all sensitive data from internet-accessible places and allocate that data to trusted devices with strong guarantees against spying. We propose the following mode of operation: Bob (Mary as well) carries around a smart USB key (a tamper-resistant token with a processor, a small secured RAM and a large persistent store) containing the private data. When the key is plugged in, Bob can issue SQL queries that link private and public data and produce aggregated outputs. Query processing algorithms on the key manage query execution on both the key and the more powerful personal computer to which the key is attached. The algorithms ensure that private detailed data never leaves Bob's key, though public data may enter the key and public aggregates (according to the selected policy) may leave the key.

Whereas Bob works in an obviously untrusted environment, most people who handle sensitive data do so as well. The availability of spyware, the uncertain incentives of system administrators, and the internet make data leaks from general purpose computers all too likely. By controlling the computing environment and the direction of information flow, GhostDB architecture provides a mechanism to ensure that those with a legitimate need to know private data are the only ones who see it.

Unfortunately, the security of the smart USB key, and thus, the security of the whole GhostDB approach is obtained at the price of strong hardware constraints. The security problem thus translates to a severe performance problem that can be overcome only with the help of special indexing and query processing techniques.

In previous work [4], we showed how OLTP-style queries (involving Select-Join-Project operators) over private and public data can be executed without leaks thanks to a smart USB key. As illustrated by the Bob and Mary scenarios, the privacy concern when producing aggregated values to a data warehouse is a burning issue too. This introduces a new and difficult challenge, namely how to compute aggregate queries over a combination of private and public data considering the limited hardware resources of the smart USB key.

The main constraints are the little RAM of the device and the read/write characteristics of the NAND Flash which both entail a complete rethinking of the indexing model, operator's algorithms and query execution techniques. It also entails novel distributed processing techniques on extremely unequal devices (standard computer and smart USB key). This paper extends [4] by introducing special algorithms to accommodate the controlled data warehousing scenario we have described above. The challenge here is more to perform these data warehousing computations in reasonable time but not necessarily to fulfill OLTP performance requirements. For these reasons, this paper introduces a rather unusual way of thinking about data warehouse architecture and physical design as well as building upon the techniques we used in the OLTP setting.

The paper is organized as follows. Section 2 sketches the mode of operation to push aggregations of a mix of private and public data to an external data warehouse. Then, given the hardware constraints of the Secure USB key, it states the problem addressed. Section 3 recalls from [4] the indexing model suggested for GhostDB and shows how to exploit it to execute Select-Project-Join queries linking Visible (public) and Hidden (private) data. Section 4 tackles aggregate computation in detail. Section 5 illustrates the combination of all operators in a query execution plan. Section 6 presents our experiments on aggregate computation and section 7 concludes.

2 Problem statement

2.1 Data Placement: Visible on Untrusted; Hidden on Secure

To clarify roles, we call the powerful but insecure general purpose storage and processing environment *Untrusted*, and the USB key *Secure*. To reflect our intended uses of the data at hand, we call the public data *Visible* and the sensitive *Hidden*. Hidden data are assumed to be downloaded to Secure through a secure channel.

Specifying which data is Visible and which is Hidden occurs at the schema definition stage. All data is by default Visible. In the create table statement, either entire tables or entire columns may be declared Hidden. For example, in a patient database, the patient primary key, age and city may be Visible, but the patient's name and body mass index should be Hidden. This is expressed simply as follows:

```
CREATE TABLE Patients (id int, name char(200) HIDDEN, age int,
                       curhosp char(200) HIDDEN, city char(100),
                       bodymassindex float HIDDEN)
```

The declaration of Hidden attributes in a table leads to a vertical partitioning of this table between Untrusted and Secure with primary keys replicated on both sides.

In practice, a large part of the database can be Visible without compromising sensitive data. For example, a design guideline could be to declare as Hidden the foreign key attributes of all tables as well as attributes whose combination could be used to identify individuals (i.e., quasi-identifiers) and let the rest of the tables and attributes remain Visible. The primary technical problem addressed in this paper concerns query processing.

Figure 1 illustrates the architecture and mode of operation of GhostDB. Queries are issued on the personal computer and transmitted as a whole to the Secure USB key. They are

expressed in SQL as usual. Depending on the query, a portion of Visible data is then requested by the PC (Visible data are assumed to be stored on remote server(s)) and enters the Secure USB key. All executions involving Hidden data or the combination of Hidden and Visible data occur on the Secure USB key, including aggregate computation. Neither Hidden data nor intermediate results ever leave that device in the clear. Final authorized aggregated data are delivered to a data warehouse for further usage.

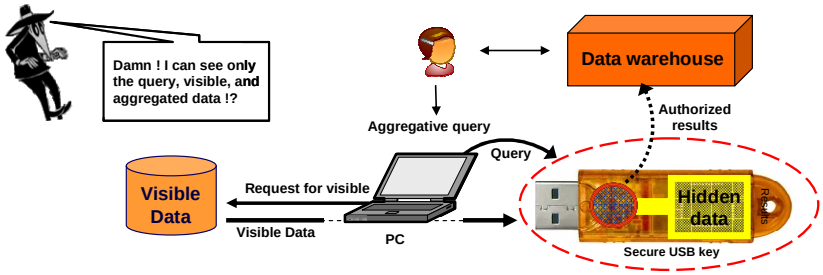


Fig. 1 GhostDB architecture and mode of operation.

While this strategy induces the transmission of a potentially large portion of Visible data, it guarantees that no Hidden data can be inferred by observing the transferred Visible data. Indeed, that portion is determined only by the query (supposed to be Visible). For example:

SELECT age, city, avg(bodymassindex) FROM Patients WHERE age>18 and CurrHosp = 'Lariboisière' GROUP BY age, city

would entail a query on Untrusted based on age that delivers a list of IDs to Secure. Secure will intersect that list with the IDs generated from the CurrHosp selection. Finally, the aggregation calculus will take place on Secure.

2.2 Hardware constraints

Secure acquires its tamper resistance from a secure chip. Secure chips appear today in a wide variety of form factors ranging from smart cards to chips embedded in smart phones and various forms of pluggable secure tokens [19]. Whatever the form factor, secure chips share several hardware commonalities. They are typically equipped with a 32 bit RISC processor (clocked at about 50 MHz), memory modules composed of ROM, static RAM (tens of KB) and a small quantity of internal stable storage and security modules. Security factors imply that the RAM must be small – the smaller the silicon die, the most difficult it is to snoop or tamper with processing. In this paper, we consider a form factor combining a secure chip with a large external Flash memory (Gigabyte sized) on a USB key having a USB2.0 full speed[4] communication throughput [28]. Figure 2 illustrates this architecture.

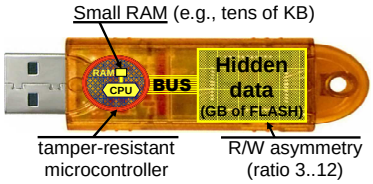


Fig. 2 Secure Computing Environment is a smart USB key.

[4] The USB2.0 full speed reaches 12Mb/s. USB2.0 High speed (up to 480 Mb/s) is envisioned for future platforms to cope with applications like on-the-fly video decryption [28].

2.3 Problem formulation

The hardware constraints of the secure USB key transform the security problem into a severe performance problem. The first challenge is to support complex SQL queries (concentrating here on Select-Project-Join-Group-By queries) with acceptable performance even for very large databases. The second challenge is to mix Visible and Hidden computations without information leaks. To handle these two problems, we adhere to the following three design rules:

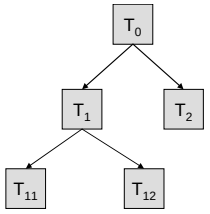
- Ensure that query processing techniques respect the fact that little RAM is available.
- Minimize the impact of irrelevant Visible data on Secure processing. Irrelevant Visible data cannot be filtered before reaching Secure without revealing Hidden information. These data must be filtered out very quickly to avoid slowing down processing on Secure.
- Prefer reads to writes based on the Flash write/read cost ratio. In Flash, writes are between 3 to 12 times slower than reads depending on the portion of the page to be read (full page vs. single word).

3 Computing selections and joins

We consider queries involving exact match and/or range selections followed by equi-joins between key and foreign key attributes over a database schema, organized as a tree (see Figure 3)[5]. We use the term *Root table* to refer to the largest central table of a database and *Node table* to refer to all non-root tables connected to the root table through direct or transitive joins on keys. The Root table is denoted by T_0 and Node tables are denoted by T_i with $i \neq 0$, where the subscript represents the position of the table in the schema, as pictured in Figure 3. The notation v_u (resp. h_u) denotes the u^{th} Visible (resp. Hidden) attribute of a table. Finally, *id* refers to the surrogate attribute of a table[6] and fk_i refers to a foreign key referencing table T_i. Using this notation, the SPJ queries of interest can be expressed as:

General form:

SELECT {T_i.id}
FROM {T_i}
WHERE {$T_i.fk_j = T_j$.id} and {$T_i.v_u \theta value_m$} and
{$T_i.h_v \theta value_n$}



Tree-structured Schema

Example:

```
SELECT   D.id, P.id, M.id
FROM     Measurements M,Doctors D, Patients P
WHERE    M.pid = P.id and P.did = D.id          // foreign keys are Hidden
         and D.specialty = 'Psychiatrist'       // Visible
         and P.bodymassindex > 25               // Hidden
```

Fig. 3 Database schema and generic select-join-project-on-key queries.

3.1 The case for a fully indexed model

While selections can always be executed in linear time in the size of a table, join performance is highly sensitive to the respective size of its operands. Join algorithms can be split in two classes depending on whether they exploit a pre-computed access structure (e.g., join index,

[5] Considering nested queries, non-equijoins or non-tree structured database schemas is left for future works.

[6] By convention, T.id refers to the surrogate attribute of a table T, id_T refers to the instances of this attribute and ID or IDs refers to the term tuple identifier(s).

bitmap index) or not. The main representatives of the latter class, also named "last resort" algorithms [10], are nested block join, sort-merge join, simple hash join, Grace hash join, hybrid hash join. An extensive performance evaluation of these algorithms can be found in [16] and shows that their performance quickly deteriorates when the smallest join argument exceeds the size of RAM. In addition, most algorithms produce intermediate results, an unfavorable situation in Flash where writes are far more costly than reads. Join indices [37] alleviate the problem. However, consecutive joins (e.g, $\sigma(T_1) \bowtie T_2 \bowtie T_3$) either incur random accesses in the join index JI_{T2T3} or a sort of the $\sigma(T_1) \bowtie T_2$ result on the IDs of T_2, a costly operation when little RAM is available and writes are expensive. Jive join and Slam join have been proposed to optimize joins through join indices [23] but both algorithms require that the number of RAM pages is of the order of the square root of the number of pages of the smaller table. The size constraint thus disqualifies these algorithms for us.

More radical solutions have been devised for the Data Warehouse (DW) context to deal with Star queries involving very large Fact tables (hundreds of GB). DW systems usually index the Fact table (i.e., root table for us) on all its foreign keys to precompute the joins with all Dimension tables (i.e., node table for us); in addition, all Dimension attributes participating in queries are indexed [22], [26], [38]. This massive indexation scheme is well adapted to the DW context where the performance of complex queries is the main issue and the update cost is not a concern.

Though GhostDB is a front-end to a DW system and does not support OLAP computations directly, it shares the same query performance issue (but must deal with more general queries than Star queries) and absence of concern for updates. In addition, the tiny RAM at our disposal in GhostDB dictates a fully indexed model. We present an indexing data structure first and then we show how to use it to combine Untrusted and Secure computations.

3.2 Subtree Key Table and Climbing Index

The primary requirement of the GhostDB indexing model is to precompute all select and join operations to minimize RAM usage. This leads to the definition of a new index structure pictured in Figure 4 for the database schema of Figure 3.

Multidimensional join indexes, as suggested in the DW context for Star schemas [26], are less RAM demanding than binary join indices [37] since combinations of joins are precomputed. To support any form of foreign key-based join expression, we introduce a data structure called the Subtree Key Table (SKT). For the root table, each tuple of SKT_{T0} concatenates the IDs of tuples from all descendant tables, thus precomputing the join with all of them. Similarly SKT_{T1} is a multidimensional join index for tables T_1, T_{11} and T_{12}.

Selection indexes could be implemented as traditional B^+-Trees. However, the processing of an expression of the form $\sigma_{hj\theta value}T_i \bowtie T_0$ would incur: (1) a lookup in T_i.hj index to get the IDs of T_i tuples satisfying the selection qualification then (2) for each of these IDs, a lookup in the $T_0.fk_i$ index to get the IDs of T_0 tuples satisfying the join expression. The final result is the union of all lists of IDs from T_0 obtained in step (2). Depending on the selectivity of the selection, the number of lists participating in the union may be large, requiring multiple passes and intermediate writes in a system with little RAM. An alternative solution may be to use bitmaps in place of lists of IDs [26], [39] . This solution decreases the cost of union but applies only to attributes on low cardinality domains, so lacks generality. Instead, we propose an index that we call a *climbing index*. A climbing index defined on an attribute contains, for each entry, one sublist of IDs per ancestor table up to the root. For example, each entry of any index on T_{12} contains a sublist of IDs for the table T_{12} itself, a sublist for the ID of T_1 and a sublist for the ID of T_0. Hence, the cost of cascading index lookups (index traversal and

union of ID lists) is avoided. Combined together, SKTs and climbing indexes allow selecting tuples in any table, reaching any non-leaf node table (including the root table) in a single step and projecting attributes from any other table. This benefit in terms of performance and RAM usage comes at an extra cost in terms of stable storage.

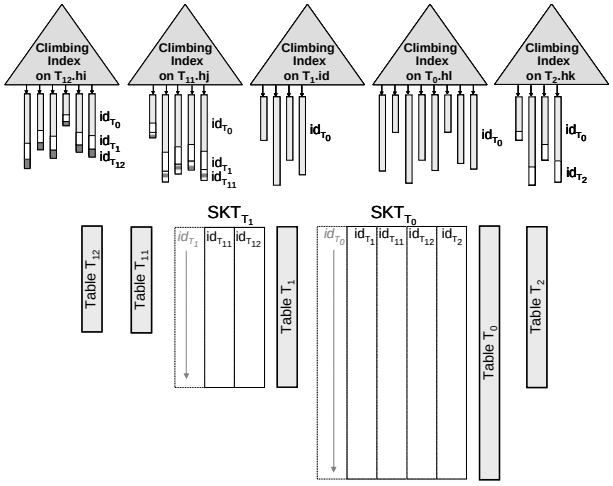


Fig. 4 Subtree Key Table and Climbing Index.

3.3 Mixing Visible and Hidden computations

Because Untrusted is fast, we want Untrusted to do as much work as possible. Under the assumption that foreign keys are Hidden[7], Untrusted is granted permission to: (1) compute Visible predicates of a query Q (i.e., select expressed on Visible attributes), (2) project the result of this computation on any Visible column, and (3) send the result to Secure.

A naïve strategy that prevents information leak would be to ask Secure to perform all the selections and joins on Hidden attributes and to perform a final join with the result of the Visible selections performed by Untrusted. The drawback to this strategy is to push the Visible selections after Hidden joins and to do the final join with a last resort algorithm. The climbing property of the climbing indexes along with the SKT provides a set of opportunities to build a much better Query Execution Plan (QEP): pushing selections before joins and performing all joins by index.

If, however, the selectivity of a Visible selection is rather low, traversing the indexes may be a poor choice. An alternative is pushing such selections after the Hidden joins by a filtering mechanism. This alternative is effective if this filtering can be done in a single pass over the result of the Hidden joins. To meet this requirement, we use Bloom filters. The Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set [8]. A bit vector is built in RAM and independent hash functions are applied to each element of the set. The false positive rate can be kept very low (e.g., less than 3%) if the size of the bit vector is at least 8 times the cardinality of the set. This property makes Bloom filters well suited to RAM-constrained environments as discussed in more

[7] This assumption could be relaxed to allow Untrusted to perform joins on Visible attributes, thus making the computation easier and more efficient. We concentrate in this paper on the most difficult situation.

detail in section 3.4. When a Bloom filter is used to filter out tuples produced by Hidden joins, false positives must be discarded at projection time by an exact selection.

Query Execution Primitives

To help explain the variety of QEPs which can be produced by combining climbing indexes, subtree key tables, and Bloom filters, we introduce the following operators:

- *Vis(Q, T, π) → {<id_T, vi_value, vj_value …>}$^{\downarrow}$:* Secure gets from Untrusted the list of IDs of Table T corresponding to tuples satisfying all Visible predicates of query Q along with attribute values for the attributes in π. Superscript $^{\downarrow}$ indicates that the list returned is sorted on the first attribute (id_T).
- *CI(I, P, π) → {{id_T}$^{\downarrow}$}:* looks up in the climbing index I and, for each entry satisfying predicate P, delivers the list of IDs referencing the table selected by π. Predicate P is of the form (attribute θ value) or (attribute ∈ {value}).
- *Merge($\cap^i${$\cup^j${id_T}$^{\downarrow}$}) → {id_T}$^{\downarrow}$:* performs the unions and intersections of a collection of sorted lists of IDs of the same table T translating a logical expression over T.
- *SJoin({id_T}, SKT_T, π) → {< id_T, id_{Ti}, id_{Tj} … >}$^{\downarrow}$:* performs a key semi-join between a list of IDs of a table T and SKT_T table and projects the result on the subset of SKT_T attributes selected by π. The result is sorted on id_T.
- *BuildBF({id_T}) → BF:* builds a Bloom filter from a list of IDs.
- *ProbeBF(BF, {< id_T, id_{Ti}, id_{Tj} … >}) → {< id_T, id_{Ti}, id_{Tj} … >}*: filters tuples from an input set using a Bloom filter.

Let us first consider a simple query involving a selection on one Visible and one Hidden attribute of the same node table, as well as a join with the root table.

Q: SELECT T_0.id FROM T_0, T_1WHERE $T_0.fk_1$=T_1.id and $T_1.v_1$θ$value_1$ and $T_1.h_1$θ$value_2$

Let us denote by QEP^{SJ} the part of a QEP dedicated to the execution of selections and joins. The simplest QEP^{SJ} for query Q would be:

1. use the index on $T_1.h_1$ in order to get sorted lists of id_{T0} resulting from $\sigma_{h1\theta value2}(T_1)$,
2. get from Untrusted the list of id_{T1} result of $\sigma_{v1\theta value1}(T_1)$,
3. for each of these id_{T1}, do a lookup on the T_1.id index to get a sorted list of id_{T0} and
4. compute the intersection between, (1) the union of the id_{T0} lists from step 1, and (2) the union of the id_{T0} lists from step 3.

This plan executing selections first, it is called Pre-Filter QEP^{SJ} (see below). Pre-Filtering suffers from the same drawbacks as cascading indexes. First, it incurs as many lookups on the T_1.id index as there are tuples resulting from the Visible selection. Second, these repetitive lookups may produce a large number of ID lists which need to be merged, a multi-pass/write-intensive process on a tiny RAM. As mentioned earlier, if the selectivity of the Visible selection is low, a post-filtering approach that pushes Visible selections after joins may outperform pre-filtering. Post-Filtering works as follows (see below):

Pre-Filter QEP^{SJ}:	*Post-Filter QEP^{SJ}:*
CI($T_1.h_1$, θ $value_2$, T_0.id) → {L_i}	BuildBF(Vis(Q, T_1, T_1.id))) → BF
CI(T_1.id, ∈ Vis(Q, T_1, T_1.id), T_0.id) →{L_j}	CI($T_1.h_1$, θ $value_2$, T_0.id) → {L_i}
Merge((∪^{i}L_i) ∩ (∪^{j}L_j)) → result	SJoin(Merge(∪^{i}L_i), SKT_{T0}, <T_0.id, T_1.id>)→F'
	ProbeBF(BF, F') → result superset

As mentioned in Section 2.3, Visible data received by Secure may include a potentially large portion of irrelevant data which cannot be filtered without revealing Hidden information. An important optimization of both Pre-Filtering and Post-Filtering is thus obtained by filtering Visible as early as possible, intersecting Visible data with the result of Hidden selections, possibly using the climbing index. Reducing Visible data cardinality

benefits Pre-Filter plans by decreasing the number of accesses to the climbing index, simplifying also the subsequent Merge phase. For Post-Filter plans, it reduces the Bloom filter size resulting in less RAM consumption and/or better filtering efficiency. We call the resulting strategies Cross-filtering.

Cross-Pre-filter QEP^{SJ}*:*

$CI(T_1.h_1, \theta\ value_2, T_1.id) \rightarrow \{L_i\}$
$Merge((\cup^i L_i) \cap Vis(Q,T_1,T_1.id)) \rightarrow L$
$CI(T_1.id, \in L, T_0.id) \rightarrow \{L_j\}$
$Merge(\cup_j L_j) \rightarrow result$

Cross-Post-filter QEP^{SJ}*:*

$CI(T_1.h_1, \theta\ value_2, T_1.id) \rightarrow \{L_i\}$
$BuildBF(Merge((\cup^i L_i) \cap Vis(Q,T_1,T_1.id))) \rightarrow BF$
$CI\ (T_1.h_1, \theta\ value_2, T_0.id) \rightarrow \{L_j\}$
$SJoin(Merge(\cup^j L_j), SKT_{T0}, <T_0.id, T_1.id>) \rightarrow F'$
$ProbeBF(BF, F') \rightarrow$ result superset

3.4 Computing projections

The complexity of the projection operation comes from distinctive features of our architecture: (1) the set of Visible attribute values sent by Untrusted may contain many values that ultimately will not appear in the result, (2) the projection operation must discard false positives generated by a Post-Filtering strategy in the result of $QEP^{SJ}(Q)$ and (3) the RAM is still a scarce resource.

To adapt to these features, the Project algorithm:

1. does the projection on a table-by-table basis to reduce RAM consumption,
2. avoids accesses to irrelevant attribute values sent by Untrusted thanks to an approximate filtering based on Bloom filters,
3. postpones the integration of all attribute values in the result tuples until the end of processing, thereby saving RAM and then iterates over the result of $QEP^{SJ}(Q)$,
4. minimizes the cost of discarding false positives.

The Project algorithm is detailed in [4] so we will not discuss it more deeply in this paper.

4 Computing Aggregates

The most natural way to compute aggregates is to perform the aggregation on Secure after the projection. This solution, called hereafter Post-Aggregation, has the advantage of working for all kinds of aggregative queries. Optimizations of this strategy are discussed in Section 4.2.

4.1 Project then aggregate on Secure

As mentioned above, the main difficulty in computing the aggregation is the little RAM available. However, two remarks merit attention. First, the last step of the projection algorithm may consume a small portion of the available RAM since all inputs are sorted on id_{T0} or equivalently on position. Thus, we consider that the result of the projection is delivered in pipeline fashion so that the largest part of the RAM is available for the aggregation. Second, the aggregation can be done in pipeline if, for each group appearing in the projection result, we can keep in RAM the grouping attribute(s) and one (or two) variable(s) to compute each aggregate (e.g., for *avg*, sum and count are kept). In this optimistic case, the aggregation cost is negligible since it does not incur any additional read or write operation in Flash memory. The algorithm is straightforward: for each incoming tuple, project the grouping attribute(s), if the associated group is already in RAM, update the aggregate value(s), or initialize a new group in RAM otherwise.

When the computation cannot be done in a single step (all groups do not fit in RAM together), different algorithms can be devised based on three design choices. We first discuss these design choices and then present strategies combining them differently. In the sequel, we

identify the input flow of the Aggregate operator with the output of the projection, that is the result of a query plan $QEP^{SPJ}(Q)$.

RAM usage: A first strategy is to use as much RAM as possible to compute the largest number of aggregated groups at each step. We call this strategy optimistic because it is based on the assumption that the number of steps will be very small. At the opposite extreme, a pessimistic strategy will dedicate the available RAM to reorganize the data produced by the project operator in a first step in order to ease the following aggregation steps. Hybrid strategies are possible by partitioning the RAM in two separate areas serving respectively the aggregation and the reorganization purposes.

Group computation order: The second design dimension concerns the criteria to select the groups that should be aggregated first at each step. The choices are: (1) FIFO: producing first the groups encountered first in the input flow, (2) Opportunistic: producing first the groups having the highest tuple cardinality, with the objective to minimize the number of unprocessed tuples and (3) Ordered: producing groups in a predefined order (e.g., sorted), skipping those which do not belong to the range of interest for the current iteration, thus allowing to identify unprocessed tuples with no storage overhead.

Unprocessed tuples: When the computation cannot be done in one pipelined step, the question of the management of unprocessed tuples (i.e., tuples corresponding to groups not computed at the current step) arises. Five strategies can be considered: (1) Materialize: write unprocessed tuples back in Flash and consider them in step i+1, (2) PartialAgg : perform partial aggregation before writing these tuples back in Flash thus minimizing writes and further reads, (3) Reorg: hash or sort these tuples before writing them back in Flash to contribute to the aforementioned reorganization purpose, (4) Mark: ignore unprocessed tuples at step i and use a bit vector to mark them in the input flow[8] for consideration at step i+1, this saves writes (only when the bit vector is smaller than the tuples not processed yet) at the expense of extra reads since each step scans the complete input flow, and (5) Skip : ignore unprocessed tuples at step i and rely on a group computation Ordered strategy to identify which tuples are of interest in the input flow at each iteration. Since Mark and Skip iterate on the input flow, they will have to pay the price of materializing it at the first step if $QEP^{SJP}(Q)$ is produced in pipeline.

Based on these considerations, several post-aggregation algorithms can be considered. They are summarized in Figure 5 and detailed below.

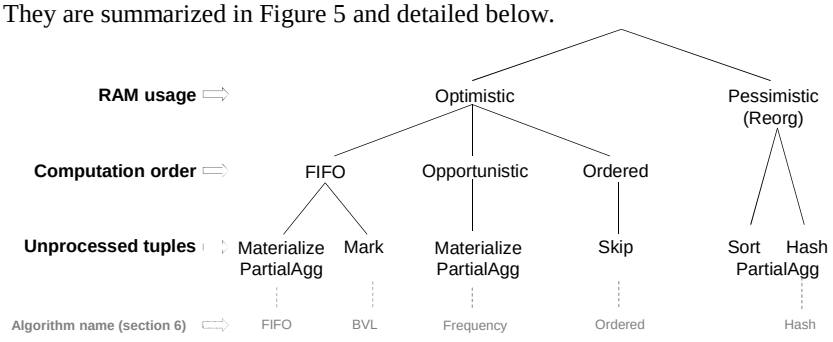


Fig. 5 Post-aggregation algorithms

Optimistic strategies are expected to outperform pessimistic ones, for input flows containing a small number of groups with respect to the size of the RAM dedicated to produce the aggregated results, because they save the initial reorganization cost. Thus, such

[8] This cannot be done in step 1 since the input flow is not materialized.

strategies, by nature, lead to minimize the RAM dedicated to store unprocessed tuples (in our settings, a RAM buffer having the size of a single Flash page is allocated), and as a consequence drastically reduce the probability, and therefore the benefit, of partial aggregation (PartialAgg).

The simplest optimistic algorithm, called *FIFO* hereafter, is a combination of optimistic RAM usage, FIFO computation order and materialization (with partial aggregation) of unprocessed tuples. Tuples that cannot be considered in the first iteration are stored in a Flash buffer OF_1. The last n groups stored in RAM which fit in a Flash page size space are written in Flash (in the buffer OF_1) and this RAM space is freed. At the end of the first iteration (when all the input flow was read), the RAM is freed and tuples from OF_1 are processed in a second iteration using the entire RAM, potentially overflowing in OF_2...OF_n. Thus, the grouping values are computed in the order where they appear in the input flow (First-In-First-Out). The main advantage of this algorithm is its simplicity.

The algorithm named *BVL* for *Bit Vector Leveling* is a variation of FIFO where the Materialize strategy for unprocessed tuples is replaced by the Mark strategy. When the RAM is going to overflow (a RAM buffer of the size of one Flash page is kept available) a bit vector is allocated in this remaining RAM space and used to mark the next input flow's tuples. When the RAM space allocated for the bit vector is full, it is flushed in a Flash page and freed to contain the next segment of the bit vector. At the next iteration, the bit vector is read when processing the input flow. It is helpful to know for each tuple if it was already processed or not. Since the groups are processed in a FIFO order, the size of the bit vector decreases at each iteration, explaining the name of the algorithm. Indeed, there is no need to build a bit vector for all tuples considered before the first RAM overflow. Note that a page of Flash can contain a mix of marked and unmarked tuples. The price to pay to consider marked tuples at the next iteration is the hardware cost to read the page from the Flash module to the Flash driver register plus the cost to transfer the relevant tuples to the RAM but the transfer cost of irrelevant tuples is saved (as shown in section 6, the gain is important).

The algorithm named *Frequency* takes advantage of biased data distributions to process most populated groups in priority, hence decreasing the number of unprocessed tuples at each iteration quicker than if the groups were processed in a FIFO order. The goal behind this is to reduce the number of tuples to write (and obviously the number of tuples to read in the next iteration). Large groups are dynamically identified by incrementing counters (the counter is a meta-data associated to each group) while doing the aggregation. When RAM overflows, the groups it contains are sorted according to their counters and the page containing groups with the smallest counters is written to Flash. The remaining groups are kept in RAM and are totally computed at this iteration. The next tuples of the input flow that do not belong to these elected groups are written to Flash (a RAM buffer, size of a Flash page, is dedicated to the tuples to flush). All iterations are similar.

The objective of the *Ordered* algorithm is to perform the complete aggregation with no write in Flash (except the initial input flow materialization, if required) and with the whole RAM for computing the aggregates. The idea is to keep in RAM at iteration i the smallest grouping values strictly greater than the last grouping value processed at iteration i-1, in the spirit of [6]. When the RAM overflows, only the smallest grouping values, which are greater than the last grouping value processed at the iteration i-1, are kept in RAM and processed. Thus, the groups are processed in an ascending order.

Pessimistic strategies are best adapted to input flows containing a large number of groups since they reduce the number of iterations dramatically. In a first iteration, the input flow is pre-aggregated in RAM and organized by hashing or sorting when written back to Flash. In our context where there is no motivation to produce a sorted result (when an order by clause exists in the query, it can be managed by Untrusted without information leak), the hashing

strategy provides better performance (obviously, hashing is less expensive than sorting). Hence we focus on *Hash* algorithm in the sequel and in the experiments. Note that writes to Flash occur on a page basis, meaning that when RAM overflows, a single page of the most populated hash bucket (the overflowing bucket) is flushed. Flushing hash buckets lazily allows maximizing the number of pre-aggregations. All the tuples corresponding to the same grouping value are located in the same hash bucket. For this reason, each hash bucket is then processed separately. Thus, the number of Flash pages read is smaller than it is in the optimistic strategies. Each hash bucket can, hopefully, be processed in a single step (otherwise, it is recursively rehashed by using another hash function).

Other combinations of strategies can be envisioned. For instance, an opportunistic group computation order could be combined with a pessimistic RAM usage to support biased distribution with a large number of tuples and groups. To this end, the available RAM needs to be partitioned between aggregate computation (for large groups) and tuple pre-aggregation and hashing (for smaller groups), thus saving writes and processing most tuples in the first iteration. Section 6 compares these different strategies through a performance analysis.

4.2 Grouping and Aggregating on Visible

Post-Aggregation strategies rely on Secure to form the groups and compute the aggregates. This section suggests better-performing alternatives enabling a better participation of Untrusted in the processing.

4.2.1 Grouping on Visible

Depending on the query, the grouping operation may be delegated to Untrusted according to the following principle. Untrusted executes the Visible part of the query and produces distinct grouping values, each associated with the list of identifiers of all Visible tuples belonging to this group. This partial result is sent to Secure which joins it with the result of $QEP^{SJ}(Q)$ on the tuple identifiers. For each matching tuple, the aggregate variable associated to the corresponding grouping value is updated. The conditions to make this possible are: (1) that the grouping attributes are Visible and (2) that the identifiers used to perform the join appear as attributes which are both Visible and Hidden.

Let us illustrate this strategy on the database schema presented in Figure 3 with a query computing an aggregation on attribute $T_0.att_0$ Group by a pair of Visible attributes $T_{11}.att_1$ and $T_{12}.att_2$. Untrusted produces all grouping pairs ($<T_{11}.att_1, T_{12}.att_2>$) along with the list of identifiers $<\{T_{11}.id\}, \{T_{12}.id\}>$ of T_{11} and T_{12} tuples satisfying all Visible predicates and belonging to this group. Secure builds a RAM structure *Buf* = $\{<T_{11}.att_1, T_{12}.att_2, \{T_{11}.id\}, \{T_{12}.id\}$, aggregate variables$>\downarrow\}$ to accommodate this partial result and joins it with $QEP^{SJ}(Q)$ on the following criteria: $QEP^{SJ}(Q).T_{11}.id \in \{T_{11}.id\}$ and $QEP^{SJ}(Q).T_{12}.id \in$ {T12.id}. The value of the attribute to be aggregated is obtained as in the projection algorithm.

The idea of this algorithm is attractive since it delegates the grouping phase to Untrusted, as well as the projection of all attributes of the Group by clause. However, three main problems must be solved. First, the Visible flow may contain many useless tuples since Hidden predicates cannot be evaluated by Untrusted. If foreign key attributes are hidden, even join predicates cannot be evaluated by Unstrusted so that a cartesian product must be performed to produce all possible pairs $<T_{11}.att_1, T_{12}.att_2>$. To limit this problem, a bloom filter strategy similar to the one introduced in section 3 can be used to filter out all irrelevant values sent by Untrusted and which cannot belong to $QEP^{SJ}(Q)$. Second, the RAM may not accommodate the whole structure *Buf*, in particular when several attributes are involved in the group by clause. In this case, several iterations are necessary. Different strategies can be devised to manage these iterations in the spirit of section 4.1 but with the difference that

grouping values comes from Untrusted. Third, false positives may be present, either produced during the filtering step described above or by a post-filtering $QEP^{SJ}(Q)$ plan. In the former case, false positives are automatically removed during the join operation. In the latter case however, post-filtering must be replaced by an exact post filtering for any Visible selection on an attribute which do not participate in the projection.

4.2.2 Aggregating on Visible

In addition to grouping, aggregate computations might also be performed on Untrusted in certain situations. The basic principle is as follows. As suggested above, Untrusted executes the Visible part of the query, produces distinct grouping values, each associated with the list of identifiers of all Visible tuples belonging to this group. In addition, Untrusted computes its own perception of the aggregate value for each group, based on the selected tuples. This partial result is sent to Secure which joins it with the result of $QEP^{SJ}(Q)$ on the tuple identifiers. The additional work for Secure is to identify Visible tuples which do not match Hidden predicates and decrement accordingly the aggregate value computed by Untrusted (i.e., substract the tuple contribution to this value). The conditions to make this possible are: (1) that the attributes involved in the Group by and the aggregate clauses are all Visible, (2) that aggregate functions are cumulative (this is the case for standard SQL aggregate functions), (3) that the identifiers (primary and foreign keys) of all tables in the path from the Group by clause to the aggregate clause (in the database schema) appear as attributes which are both visible and hidden.

To illustrate this principle, let us come back to the example of Section 4.1 and consider that a Hidden selection applies e.g., on T_{12} and T_1. The contribution of tuples from these two tables selected by Untrusted according to Visible predicates, but eliminated by Secure according to Hidden predicates, must be identified and withdrawn from the corresponding aggregate value computed by Untrusted. To allow this, for each result of the form $<T_{11}.att_1$, $T_{12}.att_2$, global_agg > computed by Untrusted, partial aggregate values corresponding to sub-groups are also computed. The resulting flow sent by Untrusted to Secure is actually of the form $\{< T_{11}.att_1, T_{12}.att_2, global\_agg, \{T_{11}.id\}, \{T_{12}.id, partial\_agg_{12}, \{T_1.id, partial\_agg_1\}\}>\downarrow\}$. This structure identifies the contribution of each tuple sent by Untrusted to the aggregation, whatever its position in the join path. For instance, if tuple $t \in T_{12}$ is disqualified by Secure, its own contribution plus all those of T_1 tuples joining with t can be withdrawn easily (by computing global_agg – $partial\_agg_{12}$). If tuple $t' \in T_1$ is disqualified by Secure, only its own contribution is withdrawn.

Note that delegation to Untrusted of the grouping/aggregate computations may also be envisioned in the case of a query involving Hidden and Visible attributes in a same group by clause. The solution is a direct extension of the idea presented above but is not detailed here for the sake of conciseness.

5 Query Execution Plan

Figure 6 presents the global QEP for an abstract query involving selections, joins, projections and aggregation over Visible and Hidden attributes. Circles represent operators and edges show the composition of operators with annotations indicating the content and ordering of their input and output operands. Superimposed boxes mean that similar subtrees can be repeated in the QEP if several predicates (on different attributes) appear in the query.

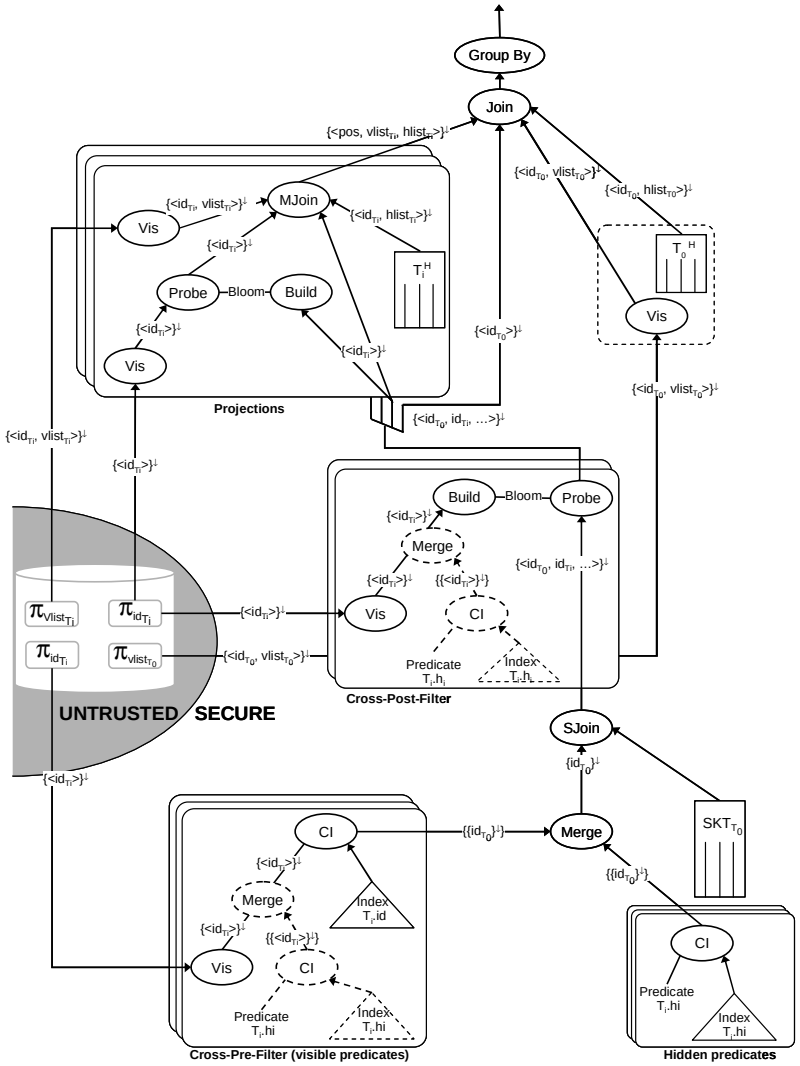


Fig. 6 Abstract Query Execution Plan for select-project-join-group-by queries on Visible and Hidden attributes.

The gray area on the left symbolizes Untrusted. For clarity, the figure shows a single table in Untrusted participating in selections on Visible attributes following either a pre-filtering strategy (bottom of the QEP) or a post-filtering strategy (middle of the QEP). The subtrees drawn in dashed lines illustrate a potential cross-filtering optimization for both strategies. The upper part of the figure highlights the particular management of projection over Visible and Hidden attributes of the root table. The QEP has been drawn considering Post-aggregation algorithms symbolized by the Group By Operator, just after the operators achieving the projection. Materialization steps are not represented because they depend on the ratio between the size of the intermediate results and the RAM allocated to the execution of each operator instance. For instance, and as we will consider in the experiments, when dealing with a large number of tuples, the RAM may be saturated during projection, thus leading to materialize the result of the upper join operator on the figure.

6 Experiments

Extensive experiments on both synthetic and real data sets have been conducted on select project-join-query (SPJ) and are presented in [4]. In this section, we focus on the evaluation of aggregate computation since this is the most critical aspect in the targeted data warehouse context. We first describe the experimental platform, the data parameters and the algorithms. We then study the impact of several parameters on the aggregation cost.

6.1 Experimental platform

Our industrial partner Gemalto has announced the availability of the first commercial smart USB keys by the end of 2008. These devices will have hardware characteristics close to the one presented in section 2.2, with 64KB of RAM and 256MB of external Flash (with a rapid growth of the Flash capacity forecast). Gemalto provided us with a software simulator for this device. Our prototype has been developed in C on top of this simulator. This simulator is not cycle-accurate so that performance measures in absolute time are not significant. However, this simulator is I/O accurate, meaning that it delivers the exact number of pages read and written in Flash. This includes the I/O performed by the Flash Translation Layer which manages wear leveling, garbage collection and translation of logical addresses to physical. The simulator delivers also the exact number of bytes transferred between the RAM and the Flash Data Register. The time to read (resp. write) k bytes in Flash and load them in RAM is composed of the time to load the page from the Flash to the Data Register in the Flash module (25µs) and the time to transfer the k bytes from the Data Register to the RAM (k×50ns). This means that reading a page costs between 25µs and 125µs depending on the portion actually loaded in RAM. Therefore, the ratio between a read and a page write in Flash roughly vary from 2.5 to 12. Other parameters are presented in Table 1.

Parameters	***Values***
Size of an ID (bytes)	4
Size of a page in Flash (bytes)	2048
RAM size (KB)	64
Time to read a page in Flash (µs)	25
Time to write a page in Flash (µs)	200
Time to transfer a byte between Data Register and RAM (ns)	50

Table 1 Main performance parameters of USB keys.

6.2 Size of the index structures

Figure 7 shows the storage cost of the SKT plus CI indexes corresponding to the database schema introduced in figure 3 (the attributes has the same default size 10 bytes).

T_0 [10 Mtuples]: (id, fk_1, fk_2, v_1^V, v_2^V, …, h_1^H, h_2^H, …)
T_1 [1 Mtuples]: (id, fk_{11}, fk_{12}, v_1^V, v_2^V, …, h_1^H, h_2^H, …)
T_2 [1 Mtuples]: (id, v_1^V, v_2^V, …, h_1^H, h_2^H, …)
T_{11} [100 Ktuples]: (id, v_1^V, v_2^V, …, h_1^H, h_2^H, …)
T_{12} [100 Ktuples]: (id, v_1^V, v_2^V, …, h_1^H, h_2^H, …)

The x-axis is the number of indexed Hidden attributes per table (in addition to primary and foreign keys), assuming for simplicity that all tables have the same number of indexed attributes. DBSize represents the total size of all Visible and Hidden raw data populating the database without indexes. The other curves represent the storage overhead incurred by the

index on Hidden data. FullIndex is the index structure presented in this paper where each non-leaf node of the database schema holds a SKT and each attribute is indexed by a climbing index referencing all ancestor tables. BasicIndex reduces the size of this index structure by considering only a single SKT (for the root table) as well as climbing indexes referencing the root table directly. The small difference between these two curves demonstrates that the extra price to pay to benefit from a complete indexation structure is low, the storage cost being dominated by SKT_{T0} and the lists of id_{T0} in all climbing indexes. StarIndex is in turn a reduction of the BasicIndex where selection indexes are traditional (i.e., contain lists of ID of the indexed table only) but include the SKT of the Root table to precompute Star joins. The large difference between BasicIndex and StarIndex shows that climbing indexes incur a significant overhead.

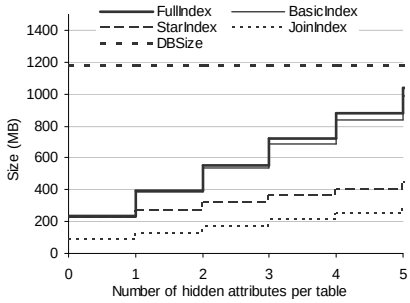


Fig.7 Storage cost of different indexing schemes.

6.3 Data and algorithms

We decided to focus on the performance of the aggregation part of queries, independently of the SPJ part already detailed in [4]. The advantage is to make easy the modification of the characteristics of the input of the aggregation operator. The results are also more general and understandable since they are independent of the rest of the query. To simplify the discussion, we assumed that the result of the SPJ query is stored on Flash but, actually, the aggregation could be computed in pipeline right after the projection.

We measured the algorithms described in Section 4.1, i.e., optimistic ones (FIFO, Frequency, Bit Vector Leveling (BVL), Ordered), pessimistic ones (Hash) and hybrids (Frequency combined with Hash).

Since the group by algorithms evaluated in the experimentations section are computed after the select, project and join operations the number of attributes in a raw data tuple, their size and their status (visible or hidden) have no impact on the execution time which was evaluated in the experimentations section. The only parameters required to know if the RAM buffer(s) will overflow is the size of each result tuple and the number of distinct groups in the flow.

The average size of the result tuples is fixed to 50 bytes. We varied alternatively three characteristics of the input data: the number of distinct groups, the distribution of the tuples on these groups and the total number of tuples to aggregate. To generate the data, we used the data generator from Jim Gray [16] that we slightly adapt to ensure that we have at least one tuple in each group even with highly biased distributions. Table 2 illustrates the characteristics of the data.

Experimental parameters	*Section 6.4*	*Section 6.5*	*Section 6.6*
Number of aggregated tuples (n)	1 Million	5 Million	**[1M to 7M]**
Number of distinct group values (N)	**[1 – 50 000]**	5000	5000
Distribution of tuples in group	Uniform	**[Uniform, Normal, Zipf]**	Uniform

Table 2 Experimental parameters.

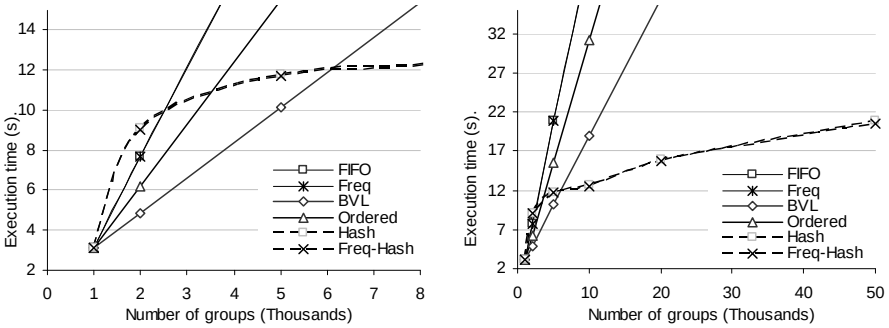


Fig. 8 Cost of post-aggregation algorithms, varying the number of distinct groups

6.4 Impact of the number of distinct groups' values

Figure 8 shows the impact of the increasing number of distinct groups' on the cost of each algorithm (with a zoom on the left for values below 8000). In our experimental setting, the small RAM can accommodate up to 1000 aggregated results. Thus, when the number of distinct group is less than 1000, all the algorithms produce the result in a single iteration with an identical cost (i.e., the cost of reading all the input tuples from Flash).

With a larger number of groups, the optimistic algorithms (FIFO, BVL, Ordered and Frequency) process 1000 groups at each iteration. They re-read, at least, at each iteration, at least the unprocessed tuples, thus explaining the bad performance when N is large.

FIFO and Frequency achieve the same performance because the distribution of the tuples in the group is uniform and because the Frequency overhead (maintaining one counter) is imperceptible. Ordered achieves better performance than FIFO and Frequency because it avoids writing unprocessed tuples (but incurs re-reading the whole input tuples at each iteration)[9]. Finally, Bit Vector Leveling algorithm is the best optimistic algorithm because it uses a bit vector to materialize the unprocessed tuples instead of writing them on Flash thus saving writes and further reads. Moreover, even if BVL reads almost the same number of pages as Ordered, it saves a lot of data transfer since it only retrieves useful tuples thanks to the bit vector.

When the number of groups is high, pessimistic and hybrid algorithms (Hash and Freq-Hash) are much better than the optimistic ones. Indeed, they reduce the number of iterations by hashing the input in several buckets and aggregate each bucket separately. BVL outperforms Hash and Freq-Hash below approximately 6500 distinct groups because the write overhead (for partitioning) is higher than the read overhead (for re-reading unprocessed tuples). Note that, as expected, Freq-Hash does not bring any significant advantage with uniform distributions.

[9] Note however that pipelined execution after the projection operator is not possible with Ordered and BVL (since they must be re-read).

To conclude this first series, optimistic algorithms with (almost) no writes (or almost no writes) like Ordered and Bit Vector Leveling have good performances when the number of groups' values is not very important (<6500). Otherwise, pessimistic algorithms are much better because they reduce drastically the number of iterations (in our setting, one iteration for partitioning and one for aggregating each bucket).

6.5 Impact of data distribution

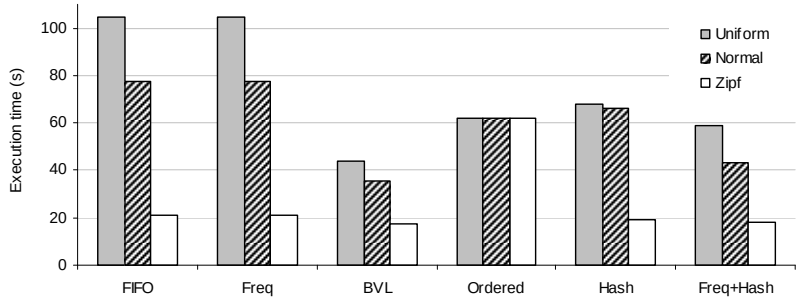


Fig. 8 Impact of the tuples distribution on groups

Figure 8 shows the cost of each algorithm with uniform, normal and Zipf distribution. With our setting, the *Normal* distribution led to concentrate 50% of the tuples in 10% of the groups. For the *Zipf* distribution, 5% of the groups have 95% of the aggregated tuples.

The figure clearly exhibits that the cost of post-aggregation algorithms is strongly linked with the distribution of tuples in the groups. Note that since the Ordered algorithm reads all the tuples at each iteration, it is not impacted by the data distribution. For the other algorithms, the high frequency of few groups (Normal and Zipf) allow aggregating a lot of tuples at a time either during aggregation (optimistic algorithms) or during partitioning (pessimistic algorithms) thus saving rewrites (FIFO, Freq, Hash and Freq-Hash) or rereads (BVL).

One could expect a more evident advantage between Freq and FIFO algorithms, since the former makes use of statistics on the data distribution to choose the first groups to be computed. Actually, the arrival order of the tuples follows a uniform law (or Zipf law), leading to compute frequent groups even without trying to detect them. Indeed, the probability of frequent groups to appear very early is very high (precisely because they contain many tuples).

The Freq-Hash algorithm may bring significant advantages compared to its Hash counterpart, especially for reasonably biased distributions (e.g. the *Normal* distribution). At RAM overflow, important write savings may be obtained, since the Freq-Hash algorithm keeps groups having a high frequency and writes Flash pages filled with groups having a low frequency (for the normal distribution in Figure 8, Freq-Hash saves 50% of writes compared with Hash). In more extreme cases of uniform and very biased distributions (see Uniform and Zipf), Freq-Hash and Hash have similar behaviors. Since the overhead of frequency-based algorithms is low, a good approach is thus interesting to use this algorithm for its robustness.

6.6 Impact of the number of aggregated tuples

Finally, we checked the scalability of the post-aggregation algorithms varying the number of tuples to aggregate (from 1 to 7 million of tuples) while keeping the number of groups

constant (5000) and considering a uniform distribution of tuples to groups. We verified that the cost of each algorithm basically increases linearly. Indeed, the number of tuples has a linear impact on the initial cost of reading (or repetitive reads for Ordered), on possible writes (bit vectors, unprocessed tuples, or hash buckets) and on subsequent reads. We did not include the figure for space consideration.

7 Conclusion

A principal function of data warehouses is to summarize detailed data into various aggregates. Such summaries frequently are derived private detailed data, even though the summary itself can be delivered to the public. If traditional computing infrastructures could be trusted, we could rely on existing database servers to host private sensitive data and produce the requested aggregations. Unfortunately, traditional security procedures on traditional infrastructures do not offer any technically solid privacy guarantees.

Trusted devices like smart USB keys allow new means to enforce data privacy, but they are very slow. This paper therefore introduces a way of designing data warehouse and physical design architectures using the secure USB key as the cornerstone for privacy preservation.

The contributions of this paper are the following: (i) we proposed and classified general aggregation algorithms enabling the efficient computation of any aggregate within the constrained hardware constraints of the GhostDB trusted device; (ii) we presented special case optimization strategies, suitable for particular visible/hidden data partitions, based on delegating the grouping and/or the aggregation work to the Visible database server; (iii) we performed experiments illustrating the behavior of these aggregation algorithms under varying data distributions, cardinalities of grouping values and base tuples.

While the algorithms were evaluated and presented separately, smart combinations, possibly including cost-based optimizations based on dynamically computed statistics (e.g., by the projection operator) may achieve the best performances and robustness. That is the direction of our future work. In the long term, we plan to work on the definition of database design tools to help select the hidden part of a database and the possibility of distributed design across a variety of smart USB key platforms.

8 References

[1] Adam, N.R, Wortmann, J.C., Security-Control Methods for Statistical Databases: A Comparative Study. ACM Comput. Survey, Vol. 21(4): *Pages* 515-556, 1989.

[2] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y., Hippocratic Databases. *The International Conference on Very Large Databases: Pages 143-154, 2002.*

[3] Agrawal, R., Srikant, R., Privacy-Preserving Data Mining. *ACM International Conference on Management of Data (SIGMOD): Pages 439-450, 2000*.

[4] Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P., Shasha, D., GhostDB: Querying Visible and Hidden data without leaks. *ACM International Conference on Management of Data (SIGMOD): Pages 677-688, 2007*.

[5] N. Anciaux, L. Bouganim, H. van Heerde, P. Pucheral, P.M.G. Apers, Data Degradation : Making Private Data Less Sensitive Over Time, *ACM Conference on Information and Knowledge Management* (CIKM), *2008*.

[6] Anciaux, N., Bouganim, L., Pucheral, P., Memory Requirements for Query Execution in Highly Constrained Devices , *The International conference on Very Large Data Bases (VLDB): Pages 694-705,* 2003.

[7] BBC News, Bank customer data sold on eBay, August 26, 2008, http://news.bbc.co.uk/2/hi/uk_news/7581540.stm

[8] Bloom, B., Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7): *Pages 422-426*, July 1970.

[9] Bolchini, C., Salice, F., Schreiber, F., Tanca, L., Logical and Physical Design Issues for Smart Card Databases. *ACM Transactions on Information Systems (TOIS)*: *Pages 254-285*, 2003.

[10] Bratbergsengen, B., Hashing Methods and Relational Algebra Operators. *The International Conference on Very Large Databases (VLDB)*, *Pages 323-333*, 1984.

[11] Computer Security Institute, CSI/FBI Computer Crime and Security Survey, http://www.gocsi.com, 2006.

[12] Computer World. NASA sites hacked. Dec. 2003. http://www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,88348,00.html

[13] Damiani, E., De Capitani Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P., Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs, *ACM Conference on Computer and Communications Security (CCS)*: *Pages 93-102*, 2003.

[14] Desai, S., Netravali, A., Thompson, M., Carbon fibers as a novel material for high-performance microelectromechanical systems (MEMS), *Journal of Micromechanics and Microengineering*, 16, 7, (2006)

[15] European Directive 95/46/EC, Protection of individuals with regard the processing of personal data, *Official Journal L 281*, 1995.

[16] Gray, J., Barkhatov, A., DBGen Synthetic Data Generator for SQL Tables and Text files on Windows Platforms, 1999. http://research.microsoft.com/~Gray/dbgen/

[17] Haas, L.M., Carey, M.J., Livny, M., Shukla, A., SEEKing the truth about ad hoc join costs, *The VLDB Journal, volume 6, number 3, Pages 241-256*, 1997.

[18] Hacigumus, H., Iyer, B., Li C., Mehrotra, S., Executing SQL over Encrypted Data in the Database-Service-Provider Model, *ACM International Conference on Management of Data (SIGMOD)*: *Pages 216-227*, 2002.

[19] Henderson, N. J., White, N. M., Hartel, P. H., iButton Enrolment and Verification Requirments for the Pressure Sequence Smart Card Biometric. *The International Conference on Research in Smart Cards*, 2001.

[20] Hillyard, D., Gauen, M. Issues around the protection or revelation of personal information. Knowledge, Technology, and Policy, 20, 2, July 2007.

[21] IBM corporation, IBM Data Encryption for IMS and DB2 Databases v. 1.1, http://www-306.ibm.com/software/data/db2imstools/html/ibmdataencryp.html, 2003.

[22] Lane, P., Oracle9i Data Warehousing Guide, Release 1 (9.0.1). Oracle Corporation, 2001.

[23] Li, Z., Ross, K.A., Fast joins using join indices. *The VLDB Journal, Vol 8, n°1, Pages 1-24*, April 1999.

[24] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M., L-Diversity: Privacy beyond K-Anonymity. *International Conference on Data Engineering (ICDE)*, 2006.

[25] Mitzenmacher, M., Compressed Bloom Filters. *ACM PODC: Pages 144-150*, 2001.

[26] O'Neil, P., Graefe, G., Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 1995.

[27] Oracle Corporation. Oracle Database, Advanced Security Administrator's Guide, 10g Release 2 (10.2). Oracle documentation B14268-02, 2005

[28] Praca, D., Next Generation Smart Card: New Features, New Architecture and System Integration, *deliverable of the Inspired IST project*, 2005.

[29] Privacy Protection Study Commission, Personal Privacy in an Information Society, *Chapter 15: The Citizen As Participant in Research and Statistical Studies*. Report transmitted to President Jimmy Carter on July 12, 1977.

[30] Pucheral, P., Bouganim, L., Valduriez, P., Bobineau, C., PicoDBMS: Scaling down Database Techniques for the Smart card, *Very Large Data Bases Journal 10(2-3): Pages 120-132*, 2001.

[31] Sullivan, B., Privacy under attack, but does anybody care? *MSNBC article*, Oct. 17, 2006.

[32] Sweeney, L., k-anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5): *Pages 557-570*, 2002.

[33] The Financial Times. Chinese military hacked into Pentagon. Sept. 2007. http://www.ft.com/cms/s/0/9dba9ba2-5a3b-11dc-9bcd-0000779fd2ac.html

[34] The Privacy Act, 5 U.S.C. § 552a, 1974. http://www.usdoj.gov/04foia/ privstat.htm.

[35] The Washington Post. Consultant Breached FBI's Computers. July 2007. http://www.washingtonpost.com/wp-dyn/content/ article/2006/07/05/AR2006070501489_pf.html

[36] UK government loses personal data on 25 million citizens. http://www.edri.org/edrigram/number5.22/personal-data-lost-uk.

[37] Valduriez, P., Join Indices, *ACM TODS, Vol. 12, No. 2, Pages 218-246*, June 1987.

[38] Vingralek, R., Gnatdb: A small-footprint, secure database system, *International Conference on Very Large Databases (VLDB)*: *Pages 884-893*, 2002.

[39] Weininger, A., Efficient execution of joins in a star schema, *ACM International Conference on Management of Data (SIGMOD)*: *Pages 542-545*, 2002.