Querying and Aggregating Visible and Hidden Data Without Leaks

Nicolas Anciaux, Mehdi Benzine, Luc Bouganim, Philippe Pucheral, Dennis Shasha

Querying and Aggregating Visible and Hidden Data Without Leaks

Nicolas Anciaux*, Mehdi Benzine*,**, Luc Bouganim*, Philippe Pucheral*,**, Dennis Shasha*,***

* INRIA Rocquencourt
Le Chesnay, France
<Fname.Lname>@inria.fr

** PRiSM Laboratory
University of Versailles, France
<Fname.Lname>@prism.uvsq.fr

*** Courant Institute of Mathematical Sciences
New York University, New York, USA
shasha@cs.nyu.edu

ABSTRACT

Imagine that you have been entrusted with private data, such as corporate product information, sensitive government information, or symptom and treatment information about hospital patients. You may want to issue queries whose result will combine private and public data, but private data must not be revealed. GhostDB is an architecture and system to achieve this. You carry private data in a smart USB key (a large Flash persistent store combined with a tamper and snoop-resistant CPU and small RAM). When the key is plugged in, you can issue queries that link private and public data and be sure that the only information revealed to a potential spy is which queries you pose. Queries linking public and private data entail novel distributed processing techniques on extremely unequal devices (standard computer and smart USB key). This paper presents the basic framework to make this all work intuitively and efficiently.

Keywords: Privacy, Secure device, Storage model.

1. INTRODUCTION

People give privacy up very easily, mostly assuming nothing can be done [23]. Patients reveal personal data for benefits such as emergency health care, only to find later that this same data ends up in insurance databases or at companies such as ChoicePoint or Intelius[1]. MySQL's 'Database in the Sky' vision is the next step toward spreading personal data in the public place. Directives and laws related to the safeguard of personal information [9], [25] slow the flow without stopping it. This 30 year old problem [21] is partly technological – private data is replicated to barely protected computers from which it finds its way through spyware or simple email to the highest bidder. The main recent change in this picture is that agencies and enterprises are now criminally liable in case of private information leaks.

Traditional security procedures do not offer the expected armor plating [6]. 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 [1] but require the database server to be trusted. Encrypted databases require either trusting the server [13], [19] or the clients [7], [11] depending on the place decryption occurs. Databases can be entirely hosted in secure hardware [4], [19], [27] but this solution applies only to very small single-user databases. Finally, an alternative solution can be anonymizing the data [16], [24] at the price of lesser data accuracy and usability.

We propose a very different approach to protect sensitive data. 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. Let us consider the following scenario. Bob is a traveling salesman and is entrusted with sensitive corporate information about customers and technical specifications. Sometimes he would like to look at his data at customer sites, on a customer computer or on his spyware-prone laptop. Bob may want to issue queries that combine public, say company's product catalog, and sensitive private information about Bob's customers and products' specifications, but he wants to be sure the sensitive data is not leaked, even if he doesn't trust the computing environment.

[1] ChoicePoint: http://www.choicepoint.com
Intelius: http://www.intelius.com

We propose the following mode of operation: Bob 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. Query processing algorithms on the key manage query execution on both the key and the powerful personal computer to which the key is attached. The algorithms ensure that private data never leaves Bob's key, though public data may enter the key.

In eventual deployment Bob needs a secure rendering platform. This could be the key itself (some smart memory sticks already hold a small LCD screen), possibly improved by technologies such as carbon fiber [8]. This could also be an external palm-style screen or tablet connected to the key or even the screen of the terminal the key is plugged into if a secure channel can be established with the video card (Digital Right Management companies are investigating this solution). Another mode of operation is sending the result to a remote secure application through a secure socket connection. Whichever the choice, the net effect is that Bob reveals to a potential spy only the queries he poses and the visible data transmitted.

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 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 approach is obtained at the price of hardware constraints (mainly a limited RAM). The privacy preservation problem thus translates to a severe performance problem that can be overcome only with the help of special storage and query processing techniques.

The principal novelties described in this paper follow directly from this challenge: (1) how to declare which data should be visible and hidden simply and how to query it, (2) how to index the data, and (3) which query processing strategies to use to link public and private data hosted on extremely unequal devices (standard computer and smart USB key). Our philosophy is to make the user's life as easy as possible (so (1) is very simple for users, database application programmers and administrators) while efficiently supporting SQL queries on arbitrarily large databases. Efficiency considerations on the small RAM Secure USB key will lead us to the design of generalized join indexes, Bloom filters for approximate filtering, the postponement of selections until after joins in certain cases, and algorithms that reflect the differences in read/write performance in the Secure USB key. Our experiments illustrate the benefits of our novel techniques on both synthetic and real data.

The paper is organized as follows. Section 2 explains how visible and hidden data are declared and queried, and giving the hardware constraints of the Secure USB key, precise the problem addressed. Section 3 introduces a new indexation model and shows how to exploit it to execute selections and joins linking visible and hidden data. Section 4 focuses on the execution of projections. Section 5 tackles aggregate computation. Section 6 illustrates the combination of all operators in a query execution plan. Section 7 presents our experiments and section 8 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 (e.g., using secure socket layer or a USB key burned by the database owner and periodically delivered to the authorized users to carry updates).

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 (we have considered but rejected more complex specifications, because ease of use is a primary goal for us). 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 are Hidden. This is expressed simply as follows:

```
CREATE TABLE Patients (
    id int, name char(200) HIDDEN,
    age int,  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. Following this guideline, one can specify a database where most Hidden data consists of keys linking Visible tuples. Then, Visible data, such as comments about treatments, reveal nothing about individuals when their relationship to identifiers and quasi-identifiers are hidden. The primary technical problem addressed in this paper concerns query processing, however, and so we reserve a full discussion of database design considerations to future work.

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. Depending on the query, a portion of Visible data is then requested by the PC (Visible data can be stored on the PC and/or 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. Neither hidden data nor intermediate results ever leave that device in the clear.

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 user query (supposed to be visible).

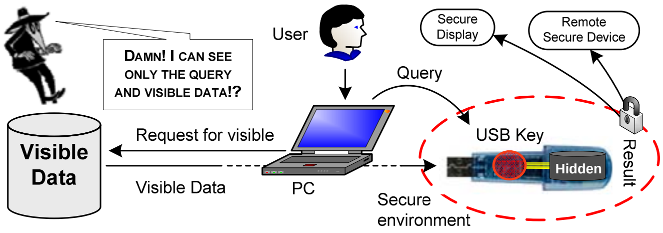


Figure 1: GhostDB architecture and mode of operation.

Queries will be expressed in SQL as usual. Queries involving only Visible attributes are executed on Untrusted with no required interaction with Secure. Queries linking Visible and Hidden data entail communication from Untrusted to Secure. For example:

```
SELECT * FROM Patients
WHERE age=50 and bodymassindex=23
```

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 bodymassindex selection. The above query is straightforward to process as are all mono-table selections. However, it incurs transferring irrelevant Visible data to Secure. This flow of irrelevant data cannot be reduced without information leakage about Hidden data.

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 [12]. 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[2] communication throughput [21]. Figure 2 illustrates this architecture.

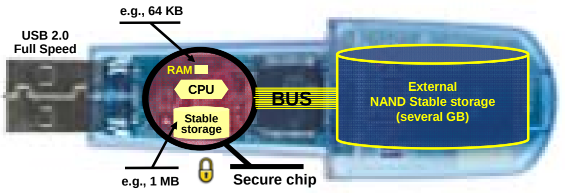


Figure 2: Secure Computing Environment is a smart USB key.

[2] 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 [21].

2.3 Problem formulation

The hardware constraints of the secure USB key transform the privacy preservation problem into a severe performance problem. Because GhostDB works in a mono-user environment on the secure USB key, simple queries (e.g., mono-table selections) and updates are of little concern provided response time can be limited to a few seconds, which is the case. The first technical challenge is to support regular SQL queries (concentrating here on Select-Project-Join queries) in order to render the performance of GhostDB acceptable even for large databases. The second technical challenge is to mix visible and hidden computations efficiently. To handle these two problems, we consider three design rules expressed below:

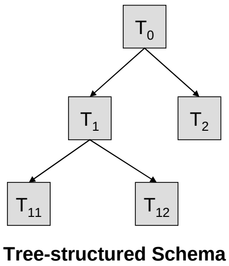
- Ensure that query processing techniques respect the fact that little RAM is available. Indeed, the device has limited RAM for security considerations. Swapping encrypted Hidden data on Untrusted could be a solution but is more expensive (encryption/decryption costs) than using the Flash memory.
- Minimize the impact of irrelevant Visible data on Secure processing. As said above, irrelevant Visible data cannot be filtered before reaching Secure without revealing Hidden information. The transfer cost is not the primary concern considering the communication throughput. However, these data must be filtered out very quickly to avoid congesting the Secure processing.
- Prefer reads to writes based on the Flash write/read cost ratio. In Flash, writes are roughly between 3 to 12 times slower than reads depending on the portion of the page to be read (full page vs. single word). Despite this discrepancy, writes on Flash are significantly more efficient than on disk (about 200µs per 2KB page).

3. COMPUTING SELECTIONS AND JOINS

This section focuses on Select-Project-Join queries involving exact match and/or range selections followed by equi-joins between key and foreign key attributes over a traditional database schema, organized as a tree (see Figure 3)[3]. 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≠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 uth Visible (resp. Hidden) attribute of a table. Finally, *id* refers to the surrogate attribute of a table[4] and fk_i refers to a foreign key referencing table T_i. Using this notation, the queries of interest can be expressed as:

General form:

```
SELECT  {Ti.id}
FROM    {Ti}
WHERE   {Ti.fkj= Tj.id} and
        {Ti.vu θ valuem} and
        {Ti.hv θ valuen}
```



Example:

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

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

For the sake of exposition, we consider projections on IDs only and delay the discussion concerning projections on non-key attributes to Section 4.

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. The TPC-C and TPC-H benchmarks give examples of database schemas and representative cardinalities. Order-line in TPC-C and LineItem in TPC-H of respective cardinalities SF×300K and SF×6M tuples (with SF a scale factor) are joined with tables roughly ten times smaller. Hence, the problem addressed in GhostDB is computing selections and joins over node tables (hundreds of thousand of tuples) and an arbitrarily large root table (millions of tuples) with a very small quantity of RAM (typically 64KB).

Join algorithms can be split in two classes depending on whether they exploit a pre-computed access

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

[4] 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).

structure (e.g., join index, bitmap index) or not. The main representatives of the latter class, also named "last resort" algorithms [5], 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 [10]. This study bears particular relevance to our context since it considers RAM sizes common a decade ago (i.e., several megabytes). That work shows that the performance of last resort algorithms quickly deteriorates when the smallest join argument exceeds the size of RAM. Except for the nested block join (which requires many passes on at least one table and has a quadratic time complexity), all algorithms produce intermediate results, an unfavorable situation in Flash where writes are far more costly than reads. Join indices [26] 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. Accessing the result tuples of the right operand table incurs random accesses or a sort. Jive join and Slam join have been proposed to optimize joins through join indices [15]. Both algorithms make a single sequential pass through each input table, in addition to one pass through the join index and two passes through a temporary file, whose size is half that of the join index. 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. In the case of a RAM size of 32×2K pages, this would imply that the smallest table could not exceed two megabytes. 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 [14], [18], [28]. 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.

Query performance is also a central issue in GhostDB and the tiny RAM at our disposal dictates a fully indexed model with the requirement to support a combination of selections and joins on both Visible and Hidden attributes. We present a new 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 in a way which minimizes 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 [18], are less RAM demanding than binary join indices [26] 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 [18], [28]. 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. For the special case of root table attributes, climbing indexes and traditional B^+-Trees are identical.

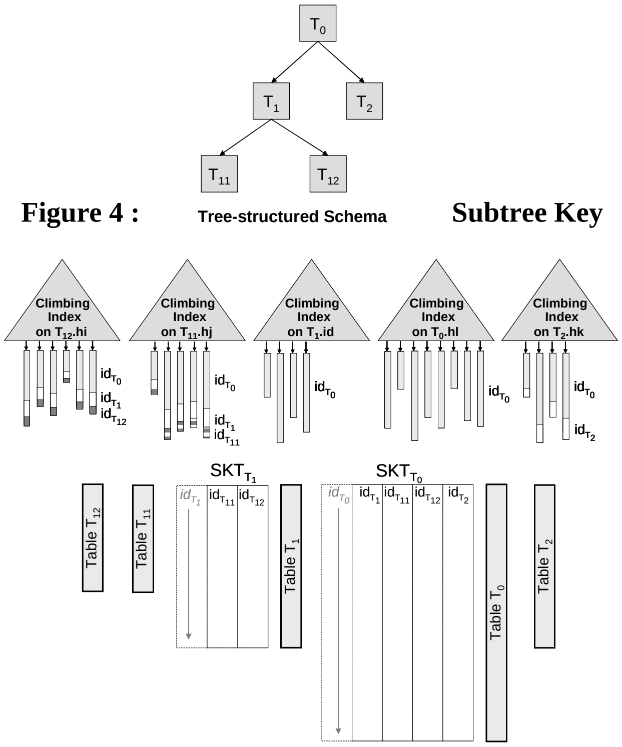


Figure 4 : Subtree Key Table and Climbing Index.

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. However, this extra cost is less than it may appear. First, the SKT columns corresponding to foreign keys come for free since they do not need to be stored in the associated table. For instance, SKT_{T1} is nothing but the projection of T_1 on all its foreign keys attributes (referencing T_{11} and T_{12}). Only the foreign keys of descendant tables other than child tables incur an extra storage cost. Second, assuming a consistent database with respect to referential constraints, SKT_T has the same cardinality as the associated table T, so that keeping the SKT_T sorted on the table identifiers of T eliminates the need to store those identifiers (e.g., the IDs of T1, pictured in grey in the figure, are not stored in STK_{T1}). Hence, the main extra storage cost is incurred by the multidimensional lists of IDs in the climbing indexes. The full set of IDs of a non-leaf node in the schema is replicated in the indexes of all its descendants. As pictured in the figure, this cost is dominated by the replication of the IDs of the root table.

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[5], 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. There is no leak of Hidden data simply because no information leaves 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. One drawback to this strategy is that it pushes the Visible selections after Hidden joins even if they are selective. A second drawback is that the strategy requires doing 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 [3]. 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 and this amount increases smoothly while the size of the bit vector decreases. This property makes Bloom filters well suited to RAM-constrained environments as discussed in more 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.

[5] 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.

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(∩^{i}{∪^{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 expressed in conjunctive normal form.

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$_1$
    WHERE T$_0$.fk$_1$=T$_1$.id and
        T$_1$.v$_1$θvalue$_1$ and
        T$_1$.h$_1$θvalue$_2$

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}.

Pre-Filter QEP^{SJ}:

CI(T$_1$.h$_1$, θ value$_2$, T$_0$.id) → {L$_i$}
CI(T$_1$.id, ∈ Vis(Q, T$_1$, T$_1$.id), T$_0$.id) → {L$_j$}
Merge((∪^{i}L$_i$) ∩ (∪^{j}L$_j$)) → result

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:

Post-Filter QEP^{SJ}:

BuildBF(Vis(Q, T$_1$, T$_1$.id))) → BF
CI(T$_1$.h$_1$, θ value$_2$, T$_0$.id) → {L$_i$}
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. Note that the redundant lookup in T$_1$.h$_1$ which appears in Cross-Post-filter QEP^{SJ} can be easily avoided in practice.

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}:

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

Let us now consider more complex queries where selections apply on Visible and Hidden attributes of different tables, followed by joins, based on hidden foreign keys.

```
Q: SELECT $T_0.id$
   FROM  $T_0, T_1, T_{12}$
   WHERE $T_0.fk_1 = T_1.id$ and $T_1.fk_{12}=T_{12}.id$ and
             $T_1.v_1 \theta value_1$ and
             $T_{12}.h_2=value_2$ and $T_0.h_3=value_3$
```

Depending on the selectivities, a pre-filtering or post-filtering approach can be selected per predicate. In addition, the Cross-(Pre or Post) filtering optimization can be exploited to combine the selectivity of selections on intermediate tables of the join tree (e.g., T_1) with the selectivity of selections on Hidden attributes of descendant tables (e.g., T_{12}). This optimization is made possible thanks to the climbing indexes which associate to each entry, one sublist of IDs per ancestor table in the join tree. Selections on the root table constitute a special and favorable case combining the selectivities of selections applied to all nodes of the join tree and delivering results sorted on id_{T0} (the ideal case for the Merge operator). We show below a candidate QEP^{SJ} where pre-filtering is selected for ($T_{12}.h_2=value_2$) and cross-post-filtering is selected for ($T_1.v_1\theta value_1$):

Mixed QEP^{SJ}:

```
$BuildBF(Merge(CI(T_{12}.h_2,=value_2,T_1.id) \cap$
    $Vis(Q,T_1,T_1.id))) \rightarrow BF$
$Merge(CI(T_{12}.h_2,=value_2,T_0.id) \cap$
    $CI(T_0.h_3,=value_3,T_0.id)) \rightarrow L$
$SJoin(L, SKT_{T0}, < T_0.id, T_1.id >) \rightarrow F'$
$ProbeBF(BF, F') \rightarrow result\ superset$
```

3.4 RAM efficient implementation of basic operators

Recall that a central requirement is to evaluate the QEP introduced above with a very small RAM (as mentioned, a typical value is 64KB, that is 32 buffers of 2KB, the I/O unit with the Flash module). Here we discuss the performance of the operators in terms of I/O, neglecting the CPU cost. Each operator has different requirements in terms of RAM. Regarding Vis, a specific buffer is dedicated to the communication channel in the smart USB key so that the download from Untrusted to Secure can be processed with no RAM consumption. All indexes in CI are implemented by means of B^+-Trees, so that CI requires at most one buffer per B^+-Tree level. SJoin implements a key semi-join between a list of IDs and a SKT sorted on the same criteria. SJoin requires only two buffers to sequentially scan the operands and one buffer to write the result.

Bloom filters consume significant RAM. The accuracy of a Bloom filter depends on the ratio m/n where m is the size of the bit vector and n is the cardinality of the set. m=8n is considered a good tradeoff between accuracy and space usage (false positive rate = 0.024 with 4 hash functions) [3]. Hence, a Bloom filter built over a list of IDs is four times smaller than the initial list, making it a good candidate to participate in RAM bounded QEP[6]. When the lists of IDs are not too large, the RAM can accommodate several Bloom filters, which can then execute in parallel on the same operand, a significant optimization. When the cardinality of the ID list is larger than the RAM size in bytes (e.g., more than 64.000 elements), we decrease the ratio m/n accordingly, entailing a smooth degradation of the Bloom filter accuracy (e.g., false positive rate becomes 0.055 for m=6n).

The Merge operator is the most complex to implement in a bounded RAM. Merge must compute expressions of the form ($L_1 \cap L_2 \ldots \cap L_k$) where each L_i is a list of IDs answering a selection predicate on a single column. The number of selection predicates in a query is usually low, and so is the number of L_i. However, an equality predicate on a Visible attribute of a node

[6] Compressed Bloom filters have been proposed but the net effect of this technique is to reduce the rate of false positives with the same space occupancy rather than decreasing the bit vector size [17]. This technique does not apply to our context because RAM is required to decompress the Bloom filter.

table generates a list of IDs of that table; when these are sent to Secure, the CI operator takes this list as input and delivers, for each element of the list, one sublist of IDs of an ancestor table in the schema (e.g., CI (T_1.id, $\in$ Vis(Q, T_1, T_1.id), id_{T0}) $\rightarrow$ $\{L_i\}$). The evaluation of range predicates on Hidden attributes also delivers a set of sublists of IDs. Hence, L_i lists corresponding to range predicates on Hidden attributes or to predicates on Visible attributes are of the form ($L_{i1} \cup L_{i2} \ldots \cup L_{ij}$) and the number of sublists L_{ij} can be arbitrarily high. Because each (sub)list is sorted on the same criteria, the computation of the complete expression of unions and intersections can be performed optimally by a sequential scan of each (sub)list L_{ij} provided the RAM is large enough to accommodate one buffer per sublist plus one buffer for writing the output. If this condition does not hold, one can consider two fundamental alternatives (smarter algorithms can be devised):

1. Perform, before the Merge, a union of sublists of IDs to reduce their total number to less than or equal to the number of RAM buffers. This can be done by loading in RAM the largest number of sublists L_{ij} of the same list L_i, sorting them to form a single sublist and writing that list back in Flash. The cost of this reduction phase being linear with the size of the sublists, the smallest sublists of each list are the best candidates for reduction.
2. Implement a Merge algorithm that requires fewer buffers than the number of (sub)lists to be merged. A rough example of such algorithm is splitting each buffer into subbuffers so that one subbuffer can be allocated to the scan of each sublist. The size of a subbuffer being smaller than a I/O unit, this entails a higher cost for the scan.

The optimal alternative depends on the ratio between the number of sublists to be merged, the number of RAM buffers, and on the element distribution in the lists. Space limitations prevent us from presenting an exhaustive analysis of the possibilities. Instead, we discuss the first alternative and its basic tradeoff: the number of sublists can be reduced to the number of RAM buffers or to fewer in order to save RAM. The benefit of the latter super-reduction strategy is to offer the opportunity to execute pipelined operators in RAM. For instance, if the RAM could accommodate the buffers required by the Merge and a (set of) Bloom filter(s), the operators of the sequence Merge-SJoin-ProbeBF could be pipelined without entailing the materialization in Flash of any intermediate result, yielding a high performance benefit. This observation about pipelining in RAM further motivates the use of climbing indexes to reduce the number of sublists resulting from the selections.

4. COMPUTING PROJECTIONS

Let us now consider complete queries including projections on both Visible and Hidden attributes. These attributes may or may not participate in selection predicates. In the example below, vlist and hlist denote respectively a list of Visible and Hidden attributes:

Query Q:
SELECT T_0.vlist, T_0.hlist, T_1.vlist, T_1.hlist, T_{12}.vlist, T_{12}.hlist
FROM T_0, T_1, T_{12}
WHERE $T_0.fk_1 = T_1$.id and $T_1.fk_{12}=T_{12}$.id and
$T_1.v_1\theta value_1$ and $T_{12}.h_2\theta value_2$ and
$T_0.h_3\theta value_3$

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 (see Section 2.3).
2. The projection operation must discard false positives generated by a Post-Filtering strategy in the result of $QEP^{SJ}(Q)$.
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,
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 $QEP^{SJ}(Q)$ and
4. minimizes the cost of discarding false positives. We concentrate below on the most difficult case where both Visible and Hidden attributes from the same table are projected.

The Project algorithm is given in Figure 5 and works as follows:

Partitioning the $QEP^{SJ}(Q)$ result: The first step of the algorithm (line 1) takes as input the result of $QEP^{SJ}(Q)$ evaluating the Where statement of query Q and performs a SJoin to get the IDs of all tables involved in the Select statement[7]. Because the Project algorithm considers one table at a time, the result of SJoin is vertically partitioned (one column per table ID, denoted by $QEP^{SJ}(Q).T_i.id$ in the algorithm) to avoid repetitive reads of unnecessary columns. All $QEP^{SJ}(Q).T_i.id$ columns are stored in root table ID order (i.e., $T_0.id$) and have all the same cardinality.

Approximate filtering of irrelevant values sent by Untrusted: For each node table $T_{i,\, i\neq 0}$, having at least a Visible attribute to be projected, a Bloom filter is built over $QEP^{SJ}(Q).T_i.id$ (line 3) and used (line 4) to filter out the irrelevant id_{Ti} sent by Untrusted at selection time (i.e., corresponding to tuples satisfying the Visible selection but disqualified by other predicates of the query). This step produces $\sigma_{VH}T_i.id$, the set of T_i IDs corresponding to tuples satisfying all Visible and Hidden predicates of the Where statement. Just as Bloom filters used in QEP^{SJ} introduce false positives for Visible selections, Bloom filters used in Project introduce false positives with respect to the QEP^{SJ} result.

Building tuples on a table basis: According to the Select statement, Visible attribute values (sent by Untrusted) and/or Hidden attribute values (taken directly in T_i^H, the Hidden image of T_i) are combined by the MJoin operator (line 6) into complete tuples <pos, T_i.vlist, T_i.hlist> where pos is the position of this tuple with respect to $QEP^{SJ}(Q).T_i.id$ column. The MJoin (MJoin stands for Merge, Multi-pass, Multi-join) operator works as follows. T_i.vlist, T_i.hlist and $\sigma_{VH}T_i.id$ are all sorted on id_{Ti} and can be joined by a sequential scan of each list and a simple merge. The result of this merge is stored in RAM up to the RAM capacity (minus two buffers). The two buffers are used to do a complete scan over $QEP^{SJ}(Q).T_i.id$ and to join it on T_i IDs with the elements present in RAM and write the result back in Flash. If required, this process is repeated $\|\sigma_{VH}T_i.id\|$ size(<T_i.id, T_i.vlist, T_i.hlist>)/(RAM – 4KB) times. The factor $\|\sigma_{VH}T_i.id\|$ exemplifies the benefit of filtering irrelevant values sent by Untrusted. MJoin automatically eliminates all false positives on this table, except those introduced by a join with false positive tuples from other tables.

Combining tuples from all tables: The last operation (line 7) joins all the resulting lists of tuples from all node tables together and potentially with Visible and Hidden attributes of the root table. All the operands being naturally sorted on id_{T0} (or equivalently on position), this operation is done by a sequential scan of each operand and a simple merge. This join automatically eliminates the false positives not discarded by MJoin.

```
(1)SJoin($QEP^{SJ}(Q)$,$SKT_{T0}$,
      <$T_i$.id/$\exists T_i$.attribute$\in$Q.ProjectList >)
(2) For each $T_{i,\, i\neq 0}$ / $\exists T_i$.vj$\in$Q. ProjectList
(3) BuildBF($QEP^{SJ}(Q).T_i.id$) $\rightarrow$ BF
(4) ProbeBF(BF, Vis(Q, $T_i$, $T_i$.id)) $\rightarrow$ $\sigma_{VH}T_i.id$
(5) For each $T_{i,\, i\neq 0}$/ $\exists T_i$. attribute $\in$Q. ProjectList
(6) MJoin([Vis(Q, $T_i$ , < $T_i$.id, $T_i$.vlist >), $\sigma_{VH}T_i.id$],
      <$T_i$.id, [$T_i^H$.hlist]>, $QEP^{SJ}(Q).T_i.id$)
                    $\rightarrow$ {<pos,$T_i$.vlist,$T_i$.hlist>}$^\downarrow$
(7) Join({{<pos, $T_i$.vlist, $T_i$.hlist>}$^\downarrow$}, [Vis(Q, $T_0$ ,
      < $T_0$.id, $T_0$.vlist >], < $T_0$.id, [$T_0^H$.hlist]>)
      $\rightarrow$ Final result
```

Figure 5: Project algorithm (QEP^P)

5. COMPUTING AGGREGATES

The most natural way to compute aggregates is to perform the aggregation after the projection. While this simple solution, called hereafter Post-Aggregation, works for all kinds of aggregative queries (mono or multi-table queries, aggregation on one or several attributes, etc.), considering projections and aggregations jointly may allow for several optimizations. In the following, we first describe this simple solution then we introduce possible optimizations on a sample query, representative of common aggregative queries.

5.1 Project then aggregate

The main problem of the Post-Aggregation strategy is that there is little RAM available to compute the aggregation. However, two remarks merit attention: (i) the last step of the projection algorithm (join operation) consumes a small portion of the available RAM since all inputs are sorted on id_{T0} or equivalently on position (see Section 4); (ii) the aggregation can be done in pipeline if, for each group appearing in the projection result, we can keep in the remaining RAM the grouping attribute(s) and one temporary value for each computed aggregate (except for *avg* which incurs the computation of *sum* and *count*). In this optimistic case, the aggregation cost is

[7] If QEP^{SJ} follows a Post-Filtering strategy this step is skipped because a SJoin has already been performed with the Root table to get all necessary IDs.

negligible since it does not incur any additional read or write operation in Flash memory. The algorithm is trivial: for each incoming tuple: project the grouping attribute(s), if the associated group is already in RAM, update the aggregate value(s), and otherwise initialize a new group in RAM.

When the computation cannot be done in one pipelined step, different strategies can be devised. The simplest strategy consists in storing in a Flash buffer OF_1, tuples that cannot be considered in the first iteration (i.e., associated to groups which did not fit in RAM). The RAM is then freed and tuples from OF_1 are processed in a second iteration using the entire RAM (except two buffers for input and output), potentially overflowing in OF_2, then $OF_3..OF_n$ for the next iterations.

Different optimizations exist for the management of these next iterations. At iteration i, the options are: (1) continue writing overflow tuples in OF_i, where $size(OF_i) < size(OF_{i-1})$; (2) write a bit vector BV_i indicating which tuples of OF_1 remain to be processed after iteration i; (3) keep in RAM the smallest grouping values strictly greater than the last grouping value processed at iteration i-1; this means treating the grouping values in a predefined order, skipping those which do not belong to the range of interest for the current iteration, in the spirit of [2].

An alternative to write overflow tuples could be to read them again from the aggregate input flow but this would incur a potentially large number of reads for recomputing the projection's join (see Section 4).

5.2 Aggregate then project

We introduce below optimizations that may apply to a large number of aggregate queries. We illustrate them on the following query pattern:

Query Q:
SELECT T_i.att1, SUM(T_0.att2)
FROM T_0, T_1, …., Ti
WHERE $T_0.fk_1=T_1.id$ and $T_1.fk_2=T_2.id$ and …
and $T_0.att_3\theta value_1$ and $T_1.att_4\theta value_2$ and …
GROUP BY T_i.att1

The goal is to apply the aggregation before the projection, thereby leading to strategies called Pre-Aggregation. Basically, the idea is: for each distinct value of the grouping attribute ($T_i.att_1$): to retrieve the list of IDs ($\{T_i.id\}$) referencing tuples sharing this value, to store it (potentially partially) in RAM, and to join it with the result of $QEP^{SJ}(Q)$ using T_i.id. This list can be either obtained from Untrusted (when att_1 is visible) or read from the Climbing index on $T_i.att_1$ (when att_1 is hidden). For the sample query Q, this means joining $\{<T_i.att_1, \{T_i.id\}>\downarrow\}$ with $\{<T_0.id, T_i.id>\downarrow\}$. The values of $T_0.att_2$ to be aggregated are obtained as in the projection algorithm (by accessing sequentially T_0 hidden or visible attribute). While the idea of this algorithm is attractive, several problems must be solved.

First, $\{<T_i.att_1, \{T_i.id\}>\downarrow\}$ may contain many useless T_i.id values (and even complete groups) since not all query predicates can be considered together. Building a Bloom filter with the values of T_i.id present in the result of $QEP^{SJ}(Q)$ may offer an efficient way to discard useless values. The result of this filter is stored in Flash since RAM is already filled by the Bloom filter.

Second, the RAM may not accommodate the whole structure $\{<T_i.att_1, \{T_i.id\}$, aggregated values$>\downarrow\}$. A simple optimization is to replace $T_i.att_1$ by an identifier referencing the corresponding value in the result of the preceding phase of filtering over $\{<T_i.att_1, \{T_i.id\}>\downarrow\}$. If it still does not fit in RAM several iterations will be necessary. In the first iteration, the RAM is loaded with as many entries as possible from $\{<T_i.att_1, \{T_i.id\}$, aggregated values$>\downarrow\}$. Then, for each tuple from $QEP^{SJ}(Q)$, we first retrieve the $T_0.att_2$ values to be aggregated. Since the result of $QEP^{SJ}(Q)$ is sorted on T_0.id, this can be done using only two buffers in RAM. If the associated T_i.id is found in RAM, then the value is aggregated. Otherwise, the result tuple is written in an overflow buffer on Flash (OF_1). When all the results of $QEP^{SJ}(Q)$ have been processed (either aggregated or sent to the overflow buffer), a first part of the final result has been computed and can be delivered after an efficient merge with the $T_i.att_1$ stored in Flash (both are in the same order). This process iterates until all groups are processed.

Third, false positives may be present, either produced during the filtering step of $\{<T_i.att_1, \{T_i.id\}>\downarrow\}$ described above or by a post-filtering Select-Join plan. In the former case, false positives are automatically removed during the join operation (the filtering step is indeed optional). In the latter case however, post-filtering must be precluded (or replaced by an exact post filtering) for any selection on a visible attribute of a table for which there is no projection of a visible attribute. For instance, a post-selection on $T_i.v_j$ using a Bloom filter, followed by an aggregation on a hidden attribute $T_i.h_1$, may lead to erroneous results.

While this algorithm seems intuitively more costly than the one presented in Section 5.1, it replaces the projection step and may thus lead to better global execution time. It is however restricted to certain query patterns. The study of all possible optimizations is out of the scope of this paper and will be the subject of future work.

6. GLOBAL 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. 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.

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. The main tradeoff in the RAM allocation decision is between the Merge and the Build-Probe operators in the QEP^{SJ} part of the plan, as discussed in Section 3.4. Other operators in QEP^{SJ} consume only a couple of RAM buffers. In QEP^{P}, the RAM allocation decision is simpler. Unlike Build-Probe in QEP^{SJ} where the size of the Bloom filter can be calibrated according to the size of its Visible operand (a set of T_i.id), Build-Probe in QEP^{P} builds a Bloom filter over $QEP^{SJ}(Q).T_i$.id, a list of IDs containing a variable number of duplicates (this number is linked to the distance between T_i and the root table in the schema). Due to the difficulty of estimating the number of duplicates, and since the project algorithm works on a table-by-table basis, the Bloom filter is calibrated by default to occupy the entire RAM. This is a poor choice if a better calibration would allow a pipelining of Build-Probe and MJoin, an improbable situation for the queries we are interested in.

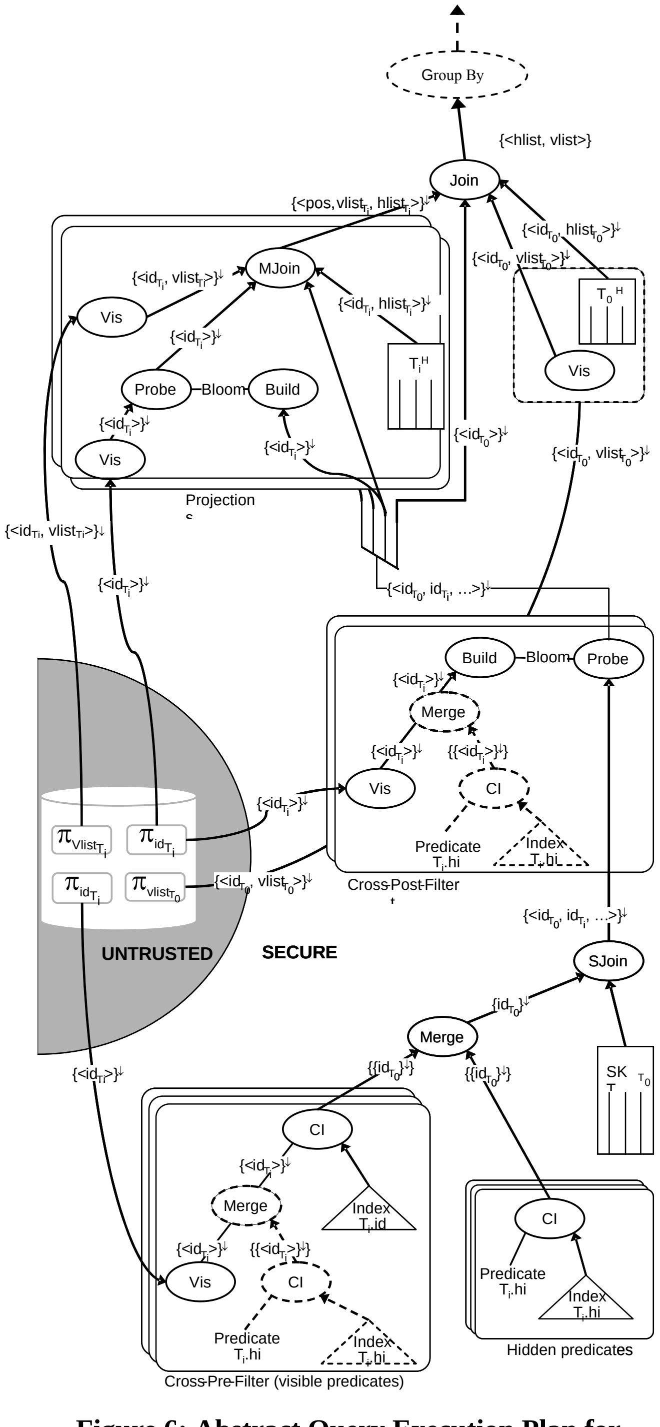


Figure 6: Abstract Query Execution Plan for general select-project-join (and group by) queries on visible and hidden attributes.

7. EXPERIMENTS

This section presents experimental results obtained from both synthetic and real data sets. We first describe the experimental platform and the data sets. The next subsections study respectively the storage overhead incurred by the proposed index, the cost of selections and joins, the cost of projections, the cost of aggregations, the impact of communication throughput and finally the cost of complete QEPs.

7.1 Experimental platform

Our industrial partner Gemalto has announced the availability of the first commercial smart USB keys by less than one year. 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 levering, garbage collection and translation of logical addresses to physical (updates are not performed in place in Flash). 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. The performance of operators is measured in terms of milliseconds, and in based on the cost of communication and I/O.

7.2 Data sets

The real dataset contains sanitized medical data related to diabetes. From this database schema, we select as hidden attribute all the foreign keys as well as attributes that could identify an individual (patients or doctors). The database schema is described below. A superscript indicates the Hidden/Visible status of the attribute; the size in bytes is indicated in brackets. As usual, primary keys are underlined while foreign keys are in italics.

Doctors [4.5 Ktuples]: (id^{VH}(4), specialty^{V}(20), description^{V}(60), first-name^{H}(20), name^{H}(20)).

Patients [14 Ktuples]: (id^{VH}(4), *doctor_id*^{H}(4), first-name^{V}(20), name^{H}(20),SSN^{H}(10), address^{H}(50), birthdate^{H}(10), bodymassindex^{H}(4), age^{V}(2), sexe^{V}(2), city^{V}(20), zipcode^{V}(6)).

Measurements [1.3 Mtuples]: (id^{VH}(4), *patient-id*^{H}(4), *Drug-id*^{H}(4), time^{V}(10), measurement^{V}(10), comment^{V}(100)).

Drugs[45tuples]: (id^{VH}(4), property^{V}(60), comment^{H}(100)).

In addition, we have built a synthetic data set to perform a comprehensive performance analysis, varying selectivities of selections on Visible and Hidden attributes (let us call them Visible and Hidden selections for simplicity). The schema of the synthetic data set is based on the schema introduced in Figure 3. In this synthetic data set, data follows a uniform distribution. Beside keys, each table has by default 5 Visible and 5 Hidden attributes each one of 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$_2^H$, …)

Parameters	*Values*
Communication throughput (MB/s)	Varying
Size of an ID (bytes)	4
Size of a page in Flash (bytes)	2048
RAM size (bytes)	65536
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.

7.3 Size of the index structures

Figure 7 shows the storage cost of the SKT plus CI indexes and compares it to possible variants. 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 is constant and 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. The advantage of FullIndex over BasicIndex is to allow for a Cross (Pre or Post) filtering optimization and the speeding up of all queries whether or not they involve the Root table. 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. StarIndex allows query execution strategies similar to [18]. The large difference between BasicIndex and StarIndex shows that climbing indexes incur a significant overhead. The next section will show, however, that these indexes yield a high performance improvement for executing selections and joins. Finally, JoinIndex is a reduction of StarIndex where the SKT of the Root table is dropped. Traditional indexes apply on all attributes, including the keys and foreign keys, allowing the execution of joins in a way similar to join indices [26]. On our real data set, the storage cost of each indexation structure is: FullIndex = 57MB, BasicIndex = 56MB, StarIndex = 36MB, JoinIndex = 26MB and DBSize = 169MB.

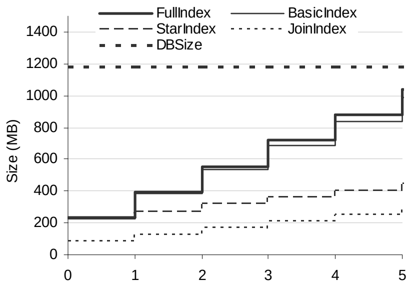


Figure 7: Storage cost of different indexing scheme.

7.4 Cost of selections and joins

This section evaluates the respective merits of the Pre and Post filtering strategies, taking advantage or not of the Cross optimization. To this end, we measure Query Q, which performs a Visible selection on T1, a Hidden selection on T12 and joins between these two tables and the Root table. We vary the selectivity of the Visible selection (denoted by sV) and fix the selectivity of the Hidden selection (denoted by sH) to 10% (sH = 0.1). In all figures, the x-axis representing sV is plotted with a logarithmic scale.

Query Q: SELECT T_0.id, $T_{1.}$id, T_{12}.id, T_1.v1
FROM T_0, T_1, T_{12}
WHERE $T_0.fk_1 = T_1.id$ and $T_1.fk_{12} = T_{12}.id$
and $T_1.v_1$ θ $value_1$
and $T_{12}.h_2$ θ $value_2$

Figure 8 shows the benefit of the Cross filtering optimization, comparing the strategies Pre-Filter with Cross-Pre-Filter and Post-Filter with Cross-Post-Filter. The figure shows that the Cross filtering optimization is beneficial whatever the selectivity of the Visible selection. The benefit becomes larger as this selectivity decreases. For sV=0.01 (resp. sV=0.5) Cross-Pre-filter outperforms Pre-Filter by a factor of 1.8 (resp. 2.3). For sV=0.5, Cross-Post-Filter also outperforms Post-Filter by a factor of 2. In this setting, the RAM can accommodate the Merge and the BuildBF-ProbeBF in parallel.

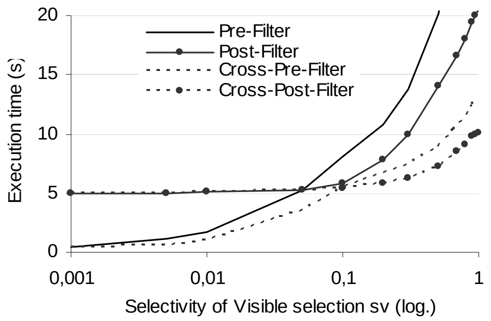


Figure 8: Filtering vs. Cross-Filtering Performance.

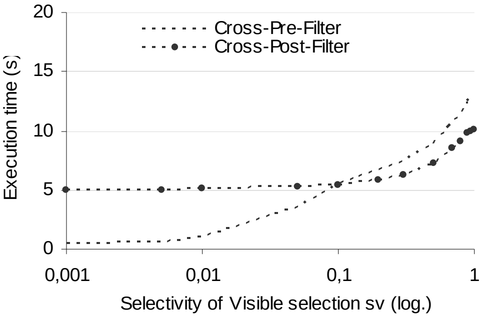


Figure 9: Cross-Pre vs. Cross-Post Filtering Performance.

Figure 9 compares the Cross-Pre-Filter and Cross-Post-Filter strategies. As expected, Cross-Pre-Filter outperforms Cross-Post-Filter when the selectivity of the Visible selection is high. Cross-Pre-Filter becomes worse for values of sV greater than 0.1. The reason is that for sV>0.1 all pages of SKT_{T0} are accessed by SJoin losing the benefit of pre-filtering. However, the differential between Cross-Pre-Filter and Cross-Post-Filter is never greater than 25%. This could lead to the conclusion that Bloom filters do not bring a significant benefit and, more generally, that postponing selections after joins (whatever their selectivity) is not very useful. This also shows that the Cross optimization is extremely effective and that indexes on Flash are far more robust than indexes on disk in low selectivity environments.

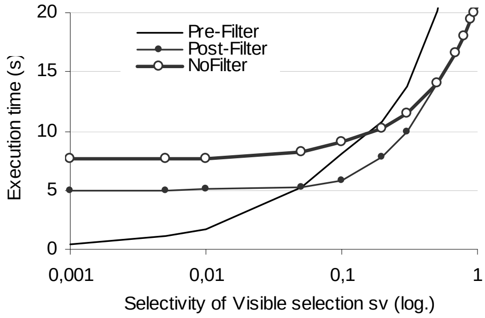


Figure 10: Pre vs. Post-Filtering Performance.

The Cross optimization can be exploited only in certain situations (more than one selection on the same table or Hidden selections on descendant tables combined with selections applied on ancestor tables). Figure 10 compares the Pre-Filter and Post-Filter strategies alone, i.e., when the Cross optimization does not apply. In this situation, the Bloom filters are much more effective. Post-Filter becomes better than Pre-Filter for values of sV higher than 0.05. For sV=0.1, Post-Filter is already 30% better than Pre-Filter. Note that the curve Post-Filter stops at sV=0.5. The reason is that for lower selectivities, the Bloom filter introduces more false positives than it can eliminate in the result of QEP^{SJ}, even if the entire RAM is allocated to the BuildBF-ProbeBF operators. In this case, Post-Filter is simply not executed and the selection is postponed to projection time. For illustrative purpose, the curve NoFilter shows the cost of postponing the selection to projection time independently of its selectivity.

To precisely capture the benefit of Bloom filters, Figure 11 compares the Post-Filter and Cross-Post-Filter strategies with Post-Select, a strategy which performs an exact selection on the result of QEP^{SJ}. Post-Select simply loads in RAM the IDs resulting from the Visible selection and filters out the result of QEP^{SJ}. Cross-Post-Select results from the Cross optimization applied to Post-Select, as usual. The figure justifies why we did not consider Post-Select as a relevant strategy in Section 3.3.

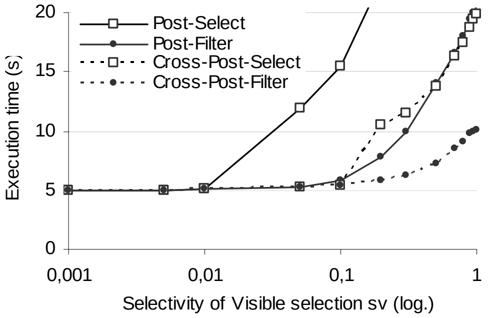


Figure 11: Post-Filtering alternatives.

7.5 Cost of projections

Figure 12 and 13 compare the cost of three projection algorithms on query Q augmented with a projection on $T_i.h_l$. Project refers to the algorithm presented in Section 4. Project-NoBF is the Project algorithm without the Bloom optimization; irrelevant attribute values sent by Untrusted are not eliminated thereby incurring a higher number of iterations. Brute-Force is an algorithm loading the result of QEP^{SJ} in RAM and accessing randomly the vlist and hlist stored in Flash. The figure shows that Project is 60% faster than Brute-Force when sV=0.1 and the gap increases with sV. Whereas Figure 12 considers a cross-pre-filtering execution, Figure 13 considers a cross-post-filtering execution to take into account the elimination of false positives introduced by the Bloom filters. Both show the insignificant impact of false positives and the effectiveness of Project algorithm.

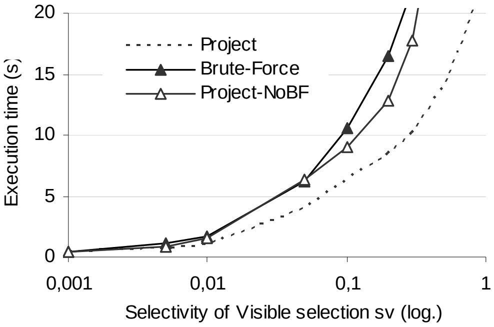


Figure 12: Projecting in Cross-Pre-Filtering execution.

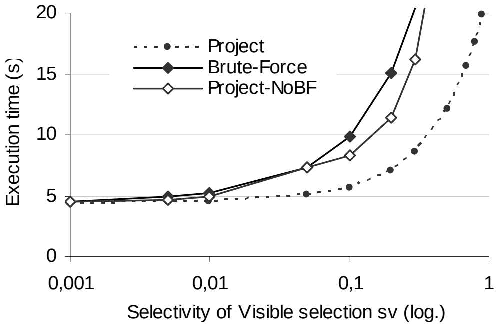


Figure 13: Projecting in Cross-Post-Filtering execution.

7.6 Cost of aggregation

Figure 14 compares the cost of the two strategies denoted Post-Aggregation and Pre-Aggregation respectively described in Section 5.1 and 5.2 on the following query:

SELECT $T_1.v_1$, SUM($T_0.h_1$) FROM T_0, T_1
WHERE $T_0.fk_1=T_1.id$ and $T_1.v_2\theta value_1$ and
$T_1.h_2\theta value_2$
GROUP BY $T_1.v_1$

The selectivity of the Visible selection (sV) is varying and the selectivity of the Hidden selection (sH=0.1) is fixed. The cardinality of attribute v1 is 1000.

For Post-Aggregation, we consider the option where overflow tuples are written in Flash at the first iteration and a bit vector is used to optimize the next iterations. For Pre-Aggregation, we consider the same option for the first iteration but the overflow buffer is read again in each subsequent iteration. Figure 14 confirms that while Pre-Aggregation is more complex than Post-Aggregation, the global execution time is reduced (around 30-40% gain). There are basically two reasons: (i) Pre-Aggregation avoids writing in Flash a substantial part of the projection result by writing only the distinct values of attribute $T_1.v_1$ rather than all instances; (ii) Pre-Aggregation projects out the $T_1.v_1$ values and keeps only in RAM the T_1 IDs and is thus almost insensitive to $T_1.v_1$ attribute size.

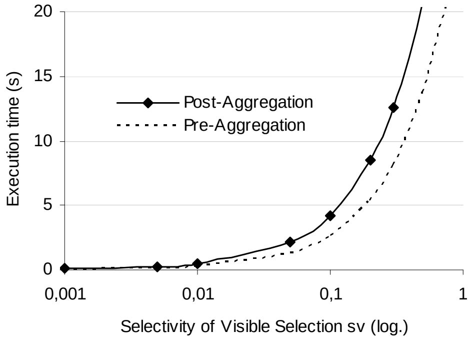


Figure 14: Projection and Group by cost

Figure 15 illustrates the latter remark and shows the impact of the grouping attribute size ($T_1.v_1$) on the cost of the aggregation operation. The selectivity of the Visible and Hidden selections are fixed (sV = 0.1 and sH= 0.1). As expected, the cost of the Post-Aggregation increases with the size of the grouping attribute while the cost of Pre-Aggregation stays rather constant. Remark that projecting out the grouping attribute in the Post-aggregation is not feasible since the value of the grouping attribute is used to associate the tuples to the correct group; moreover the values of the grouping attribute are unordered in the projection results making any substitution (for instance using hash without collision) very expensive.

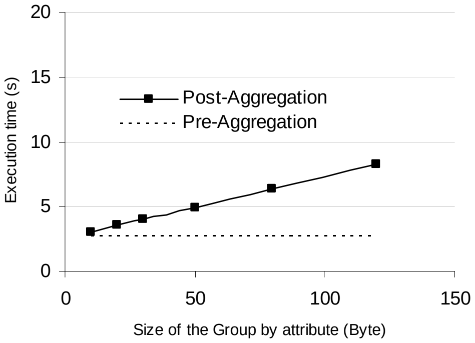


Figure 15: Impact of the group by attribute size

7.7 Communication costs

Figure 16 shows how the communication throughput impacts the global performance of a query. The x-axis represents the throughput expressed in MBps ranging from 300KBps up to 10MBps. The query measured is the same as before, except that it projects the result on one (Project1), two (Project2) or three (Project3) Visible attributes of 10 bytes each. The query is executed following a Cross-Pre-Filtering approach considering a selectivity sV=0.01. The curves highlight the fact that, for this query, a communication throughput lesser than 1.3MBps becomes the main bottleneck.

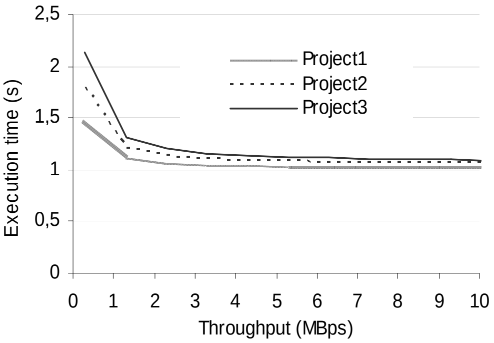


Figure 16: Impact of the communication throughput.

7.8 Query cost decomposition

The histograms presented in Figure 17 show how the total execution time (excluding the communication time) is decomposed among the operators involved in the execution of the query Q. Only the most time-consuming operators are visible in the figure, namely, Merge, SJoin and Project. Store represents the time to materialize the intermediate result produced by SJoin (in low selectivity queries). PRE (resp. POST) 1, 5 and 20 denote a Cross-Pre-Filtering strategy (resp. Cross-Post-Filtering strategy) for a respective selectivity of the Visible selection of sV=0.01, sV=0.05 and sV=0.2. PRE is shown better than POST for sV=0.01, and sV=0.05 but becomes worse for sV=0.20. As already mentioned, for sV>0.1 all pages of SKT_{TO} are accessed by SJoin losing the benefit of Pre-Filtering. Hence, the SJoin cost is the same in PRE20 and POST20 while the Merge cost is much higher in PRE20 than in POST20.

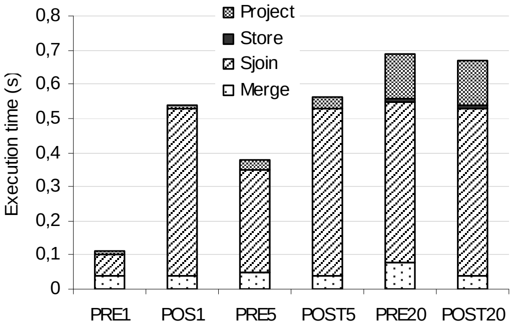


Figure 17: Cost decomposition for query Q on the synthetic dataset.

Figure 18 presents the same analysis on the real data set. Query Q keeps the same structure, replacing table T_0 by Measurements, table T_1 by Patients and table T_{12} by Doctors. The main difference between the synthetic data set and the real one is the size of the Root table (10M tuples in T_0 compared to 1.3M tuples in Measurements) and the ratio T_0/T_1 (T_0/T_1 = 10 compared to Measurements/Patients $\approx$ 92). The first observation is that the execution time of the query is related to the size of the Root table and is roughly 1/10 the times of the synthetic data set. The second observation is that the cost of the SJoin operator is dominant in all histograms. In fact, the relative cost of Project decreases because the cardinalities of the node tables (Patients and Doctors) are much smaller while the relative cost of SJoin increases due to the ratio Measurements/Patients$\approx$92.

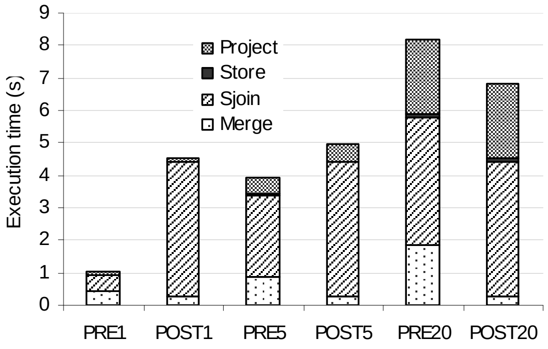


Figure 18: Cost decomposition for query Q on the real dataset.

8. CONCLUSION

People talk about privacy, but give it up very easily, especially when faced with complex security procedures that offer only conditional guarantees.

This implies that for people's sensitive data to be protected, the cost to protect it must require little physical effort and must perform well.

This paper proposes a system called GhostDB whereby people carry hidden sensitive data on a smart USB key and they plug that key into a personal computer when they need to link their hidden data with visible public data, all with the assurance that no hidden data will ever go out in the open. In terms of the administration interface, the only change is the "hidden" annotation on certain fields in the create table command. Users issue completely standard SQL, so application logic is unchanged.

GhostDB entails technical challenges having to do with distributed query processing on devices having vastly different capabilities in terms of speed, RAM size, and communication capability. We have presented a query processing framework based on the unconventional tradeoffs present in this unusual environment. The techniques show a significant improvement over naïve methods and make the solution applicable even for complex queries over large databases. In the course of this work, we have introduced techniques like climbing indexes, Subtree Key Tables and post-filtering by Bloom filters which may have wider applicability (e.g., in the Data Warehouse domain). Our future work concerns the definition of database design tools to help select the hidden part of a database, the inclusion of a cost-based optimizer, and the possibility of distributed design across a variety of smart USB key platforms.

Acknowledgments

We would like to thank Christophe Salperwyck for his help on implementing GhostDB prototype.

Shasha's work has been partly supported by the U.S. National Science Foundation under grants IIS-0414763, DBI-0445666, N2010 IOB-0519985, N2010 DBI-0519984, DBI-0421604, and MCB-0209754. This support is greatly appreciated.

9. REFERENCES

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

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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)

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

[10] 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.

[11] 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.

[12] 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: Pages 124-134*, 2001.

[13] 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.

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

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

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

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

[18] O'Neil, P., Graefe, G., Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record, Pages 8-11*, 1995.

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

[20] 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.

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

[22] 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.

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

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

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

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

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

[28] Weininger, A., Efficient execution of joins in a star schema, *ACM SIGMOD international conference on Management of data*: *Pages 542-545*, 2002.