Memory Requirements for Query Execution in Highly Constrained Devices

Nicolas Anciaux* Luc Bouganim** Philippe Pucheral*,**

* PRISM Laboratory
78035 Versailles - France
<Firstname.Lastname>@prism.uvsq.fr

** INRIA Rocquencourt
France
<Firstname.Lastname>@inria.fr

Abstract

Pervasive computing introduces data management requirements that must be tackled in a growing variety of lightweight computing devices. Personal folders on chip, networks of sensors and data hosted by autonomous mobile computers are different illustrations of the need for evaluating queries confined in hardware constrained computing devices. RAM is the most limiting factor in this context. This paper gives a thorough analysis of the RAM consumption problem and makes the following contributions. First, it proposes a query execution model that reaches a lower bound in terms of RAM consumption. Second, it devises a new form of optimization, called iteration filter, that drastically reduces the prohibitive cost incurred by the preceding model, without hurting the RAM lower bound. Third, it analyses how the preceding techniques can benefit from an incremental growth of RAM. This work paves the way for setting up co-design rules helping to calibrate the RAM resource of a hardware platform according to given application's requirements as well as to adapt an application to an existing hardware platform. To the best of our knowledge, this work is the first attempt to devise co-design rules for data centric embedded applications. We illustrate the effectiveness of our techniques through a performance evaluation.

1 Introduction

Pervasive computing is now a reality and intelligent devices flood many aspects of our everyday life. As stated by the Semiconductor Industry Association, the part of the semiconductors integrated in traditional computers represents today less than 50% of a market of $204Billion [SIA02]. As new applications appear, the need for database techniques embedded in various forms of lightweight computing devices arises. For example, the vision of the future dataspace, a physical space enhanced with digital information made available through large scale ad-hoc sensor networks is paint in [ImN02]. Sensor networks gathering weather, pollution or traffic information have motivated several recent works [MH02, BGS00]. They have brought out the need for executing local computation on the data, like aggregation, sort and top-n queries [CaK97], either to save communication bandwidth in push-based systems or to participate in distributed pull-based queries [MFH02]. Personal folders on chip constitute another motivation to execute on-board queries. Typically, smartcards are used in various applications involving personal data (such as healthcare, insurance, phone books etc.). In this context, queries can be fairly complex (i.e., they can involve selections, joins and aggregation) and their execution must be confined on the chip to prevent any disclosure of confidential data [PBV01]. Hand-held devices are other forms of autonomous mobile hosts that can be used to execute on-board queries on data downloaded before a disconnection (e.g., personal databases, diary, tourist information). Thus, saving communication costs, preserving data confidentiality and allowing disconnected activities are three different concerns that motivate the execution of on-board queries on lightweight computing devices [NRC01, Ses99].

While the computing power of lightweight devices globally evolves according to Moore's law, the discrepancy between RAM capacity and the other resources, notably CPU speed and stable storage capacity, still increases. This is especially true for Systems on Chip (SoC) [NRC01, GDM98] where RAM competes with other components on the same silicium die. Thus, the more RAM, the less stable storage, and then the less embedded data. As a consequence, SoC manufacturers privilege stable storage to the detriment of a RAM strictly calibrated to hold the execution stack required by on-board programs (typically, less than 1KB of RAM is left to the applications in smartcards even in the advance prototypes we recently experimented). This trade-off is recurrent each time the silicium die size needs to be reduced to match physical constraints such as thinness, energy consumption or tamper resistance. Another concern of manufacturers is reducing the hardware resources to their minimum in order to save production costs on large-scale markets [SIA02]. Thus, RAM will remain the critical resource in these environments and being able to calibrate it against data management requirements turns out to be a major challenge.

As far as we know, there is today no tool nor academic study helping to calibrate the RAM size of a new hardware platform to match the requirements of on-board data centric

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

applications. In traditional DBMSs, query evaluation techniques resort to swapping to overcome the RAM limitation. Unfortunately, swapping is proscribed in highly constrained devices because it incurs prohibitive write costs in electronic stable memories and it competes with the area dedicated to on-board data. In the absence of a deep understanding of RAM consumption principles, pragmatic solutions have been developed so far. Light versions of popular DBMS like Sybase Adaptive Server Anywhere [GIG01], Oracle 8i Lite [ORA02], SQLServer for Windows CE [Seg01] or DB2 Everyplace [KLL01] have been designed for hand-held devices. Their main concern is reducing the DBMS footprint by simplifying and componentizing the DBMS code [Gra98]. However, they do not address the RAM issue. Query executions exceeding the RAM capacity are simply precluded, thereby introducing strong restrictions on the query complexity and on the volume of the data that can be queried. Other studies have been conducted to scale down database techniques in the particular context of smartcards [ISO99, PBV01]. In [PBV01], we tackled the RAM issue by proposing a dedicated query evaluator relying on an ad-hoc storage and indexation model. This solution, called PicoDBMS, has been shown convenient for complex personal folders (e.g., healthcare folders) embedded in advanced smartcard platforms [ABB01]. Without denying the interest of the PicoDBMS approach, its application's domain is reduced by two factors. First, PicoDBMS makes an intensive use of indices with a side effect on the update cost and on the complexity of the transaction mechanisms enforcing update atomicity. One may thus wonder whether the resort to indices could be avoided, and in which situations. Second, PicoDBMS constitutes an ad-hoc answer to a specific hardware platform. As noticed earlier, hardware configurations are more and more specialized to cope with specific requirements in terms of lightness, battery life, security and production cost. Building an ad-hoc query evaluator for each of them will rapidly become cumbersome and, above all, will incur a prohibitive design cost [NRC01].

In the light of the preceding discussion, there is a clear need for defining pre-designed and portable database components that can be integrated in Systems on Chip (SoC). The objective is to be able to quickly differentiate or personalize systems in order to reduce their cost and their time-to-market [GDM98]. To this end, a framework for designing RAM-constrained query evaluators has to be provided. This paper precisely addresses this issue, following the three steps approach outlined below.

Devising a RAM lower bound query execution model

This study proscribes swapping and indexing for the reasons stated earlier. Searching for a RAM lower bound query execution model in this context forces us to concentrate on the algorithmic structure of each relational operator and on the way the dataflow between these operators must be organized. The contribution of this step is to propose operators' algorithms that reach a RAM lower bound and to guidelines that remain valid when a small quantity of RAM is added to the architecture.

Devising optimization techniques that do not hurt this RAM lower bound

Obviously, a RAM lower bound query execution model exhibits poor performance. The absence of swapping and indexing leads to recompute repeatedly every information that cannot be buffered in RAM. The consequence on the query execution algorithms is an inflation in the number of iterations performed on the on-board data. The contribution of this step is to propose new optimization techniques that drastically reduce the number of irrelevant tuples processed at each iteration.

Studying the impact of an incremental growth of RAM

Mastering the impact of RAM incremental growths has two major practical outcomes. In a co-design perspective, it allows to determine the minimum amount of RAM required to meet a given application's requirement. In the context of an existing hardware platform, it allows to calibrate the volume of on-board data and the maximum complexity of on-board queries that remain compatible with the amount of available RAM. As demonstrated by our performance evaluation, very small RAM growths may lead to considerable performance gains. This motivates further the study of a RAM lower bound query execution model and of RAM incremental variations. The contribution of this step is twofold. First, it proposes an adaptation of the preceding execution techniques that best exploit each RAM incremental growth and demonstrates that they constitute an accurate alternative to the index in a wide range of situations. Second, it proposes co-design guidelines helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and response time.

This paper is organized as follows. Section 2 introduces important assumptions that delimit the context of the study. Section 3 presents our RAM lower bound query execution model. Section 4 addresses optimization issues in this RAM lower bound context. Section 5 describes the impact of RAM incremental growths on the query execution model. Section 6 presents our performance evaluation. Finally, section 7 concludes the paper.

2 Context of the study

This section introduces hypothesis on the data sources, on the queries and on the computing devices, and discusses their relevance with respect to the target of this study.

H1 : On-board data sources are sequential files

We assume that the data sources are hosted by the device and do not benefit from any index structure. The reason to push indices aside from this study is threefold. First, indices have a negative side effect on the update cost (as we will see, this effect is magnified by hypothesis H4). Second, indices makes update atomicity more complex to implement [PBV01] and then have also a negative side effect on the algorithm's footprint. Finally, indices compete with on-board data on stable memory, thereby reducing the net storage capacity of the device. This does not mean that indices are definitely useless or unsuitable. As shown in Section 6, indices remain the sole solution to cope with strict response time constraints in the case of complex queries over a large amount of data. One objective - and contribution - of this study is to

demonstrate that alternatives to indices exist and are convenient in a wide range of situations.

H2 : Queries are unnested SQL queries

We consider relational data for the sake of generality and simplicity. Note that the relational model has become a standard even in highly constrained environments [ISO99,PBV01]. The queries of interest are unnested SQL queries including Group by, Having and Order by statements. Even if more complex queries could be considered, unnested SQL queries are expressive enough to cover the need of the target on-board data centric applications (sensors, personal folders, ambient intelligence).

H3 : Computing devices are autonomous

Autonomy means that the execution of on-board queries relies only on local computing resources. Obviously, if we assume that external resources can be exploited, the RAM problem vanishes. As stated in the introduction, saving communication costs, preserving data confidentiality and allowing disconnected activities are three common motivations to execute on-board queries in autonomy.

H4 : Stable storage is made of electronic memory

This assumption simply expresses the reality since lightweight computers use commonly EE-PROM, Flash or power-supplied RAM technologies for their stable storage [NCR01]. These technologies exhibit good properties in terms of lightness, robustness, energy consumption, security and production costs compared with magnetic disks. In addition, fine grain data can be read-accessed from stable storage at a very low cost (the gap between EE-PROM, Flash and RAM in terms of direct read-access time is less than an order of magnitude). On the other hand, writes in EE-PROM and Flash are extremely expensive (up to 10 ms per word in EE-PROM) and the memory cell lifetime is limited to about 10^5 write cycles.

The conjunction of H3 and H4 precludes a query evaluator resorting to memory swap. Indeed, swapping incurs prohibitive write costs and may hurt the memory lifetime depending on the stable storage technology used. But above all, the swapping area must be local and there is no way to bound it accurately. Again, the swapping area competes with the on-board data in stable storage.

3 RAM lower bound query execution model

This section concentrates on the definition of a query execution model that reaches a lower bound in terms of RAM consumption, whatever be the complexity of the query to be computed and the volume of the data it involves. The RAM consumption of a *Query Execution Plan* (QEP) corresponds to the cumulative size of the data structures used internally by all relational operators active at the same time in the QEP plus the size of the intermediate results moving along the QEP. We first present two design rules that help us to derive the principles of a *RLB* (RAM Lower Bound) query evaluator. Then, we propose data structures and algorithms implementing this query evaluator.

3.1 Design rules and consistency constraints

Two design rules guide the quest for a RLB query evaluator.

R1 (Invariability): Proscribe variable size data structures

R2 (Minimality): Never store information that could be recomputed

Rule R1 states that a data structure whose size varies with the cardinality of the queried data is incompatible with the reach of a RAM lower bound. As a consequence of R1, if an operator OP1 consumes the output of an operator OP2, OP1's consumption rate must be higher than or equal to OP2's production rate to avoid storing an unbounded flow of tuples. Rule R2 trades performance for space saving. As a consequence of R2, intermediate results are never stored since they can be recomputed from the data sources at any time. Roughly speaking, R1 rules the synchronization between the operators in the QEP while R2 minimizes the internal storage required by each of them. The conjunction of R1 and R2 draws the outline of a strict pipelined query execution model that enforces the presence of at most one materialized tuple at any time in the whole QEP, this tuple being the next one to be delivered by the query evaluator.

Let us give the intuition of such a query evaluator on an example. Let assume a Join operator combining two input sets of tuples called *ILeft* and *IRight*. *ILeft* and *IRight* result themselves from the evaluation of two sub-queries Q_{Left} and Q_{Right} in the query tree. To comply to R1, each *ILeft* tuple will be compared to *IRight* tuples, one after the other, at the rate of their production by Q_{Right}, and the result will be itself produced one tuple at a time. To comply with R2, Q_{Right} will be evaluated $|ILeft|$ times in order to save the storage cost associated to *IRight*. Following this principle for all operators and combining them in a pipelined fashion is the intuition to reach a RAM lower bound for a complete QEP. However, care must be taken on the way the dataflow is controlled to guarantee the consistency of the final result. To this end, we introduce two consistency constraints that impact the iteration process and the stopping condition of each algorithm presented in the next section

Unicity: the value of a given instance of the Cartesian product of the involved relations must be reflected at most once in the result.

Completeness: the values of each instance of the Cartesian product of the involved relations that satisfies the qualification of the query must be reflected in the result.

3.2 Execution model and notations

Before going into the details of each operator's algorithm, we discuss the way operators interact in a QEP. We consider the Iterator model [Gra93] where each operator externalizes three interfaces: *Open* to initialize its internal data structures, *Close* to free them and *Next* to produce the next tuple of its output. The QEP takes the form of a tree of operators, where nodes correspond to the operators involved in the query, leaves correspond to the Scan operator iterating on the involved relations and edges materialize the dataflow between the operators. Figure 1 pictures a simple QEP and introduces notations that will be used in the algorithm's description.

Rules R1 and R2 guarantee the minimality of the dataflow by reducing its cardinality to a single tuple. A shared data structure called *DFlow* materializes this dataflow. Each operator uses in its turn this data structure to consume its input and produce its output, one tuple at a time. More precisely,

DFlow contains: (i) one cursor maintaining the current position reached in each relation involved in the QEP; these cursors materialize the current instance of the Cartesian product of these relations; and (ii) a storage area for each attribute computed by an aggregate function. Thus, *DFlow* contains the minimum of information required to produce a result tuple at the root of the QEP and is the unique way by which the operators communicate. As pictured in Figure 1, cursors and attributes in *DFlow* are organized into lists to increase the readability of the algorithms (e.g., *GrpLst* denotes the list of cursors referencing the relations participating in the grouping condition).

Additional notations are used all along the paper. *ILeft* and *IRight* denote respectively the left and right inputs of a binary operator; |*input*| denotes the tuple cardinality of an operator's input; *k* denotes the number of distinct values in an input with respect to a grouping or a sorting condition.

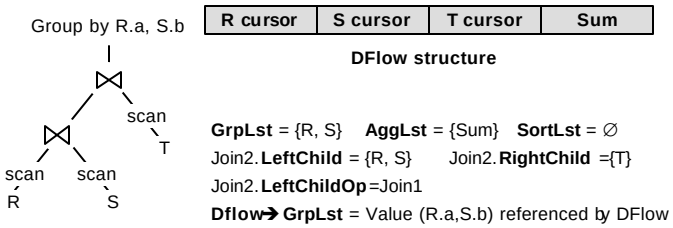

Figure 1: *Query Example and notations*

3.3 Operator's algorithms

Following the precept of rule R2, we present below a RLB form of each relational operator's algorithm. For each algorithm, we show its compliance with the Unicity and Completeness consistency constraints and give its RAM consumption.

Scan, Select, Project and Join algorithms

The algorithms implementing these operators are pictured in Figure 2, except for the Project that is rather straightforward. Project simply builds a result tuple by dereferencing the cursors present in *DFlow* and by copying the values belonging to *Dflow➔AggLst*. The Scan and Select algorithms are self-explanatory. Join implements a Cartesian product between its left and right inputs. Let us verify Unicity and Completeness for each algorithm. Regarding the Scan, the same tuple cannot be considered twice between an Open and a Close and all tuples are considered since Open initializes the cursor to the beginning of the relation and the stopping condition of Next is to reach EOF. Select and Project are unary operators that consume their input sequentially and then inherit Unicity and Completeness from their child operator in the QEP. The Join algorithm performs a nested loop on its operands so that each instance of the Cartesian product of its left and right inputs is examined exactly once. Again, Unicity and Completeness are inherited from its left and right child operators in the QEP.

The minimality of these four algorithms in terms of RAM consumption is not questionable since their consumption equals zero. Indeed, they do not use other data structure than *DFlow*.

```
Scan.Open   Dflow.Ri←0                          //  Initialize Ri scan
Scan.Next   Dflow.Ri ++                         //  reference next Ri tuple
            if Dflow.Ri > Ri.Cardinality then Dflow.Ri←EOF  //  check EOF
Select.Open ChildOp.Open()
Select.Next repeat                              //  Get Next child tuple and
              ChildOp.Next()                    //check the selection predicate
            until DFlow.Child = EOF or match(SelectionPredicate)
Join.Open   LeftOp.Open();RightOp.Open() //  Open L and R input
            if DFlow.LeftChild=EOF or DFlow.RightChild=EOF then
              DFlow.Child←EOF               //  One input is empty
            else LeftOp.Next()              //  get the first L tuple
Join.Next   if DFlow.Child ≠ EOF then       //  check both L&R EOF
              Repeat                  // Get a R tuple until EOF or match
                RightOp.Next()
                if DFlow.RightChild = EOF then
                  LeftOp.Next()         //if end of R, get a next L tuple
                  if DFlow.LeftChild ≠ EOF then
                    RightOp.Open()      //  and reopens R
                    RightOp.Next()
              until DFlow.Child = EOF or match(JoinPredicate)
GBY.Open    ChildOp.Open()                  //  Scan the whole input in
            Split.Value←+∞                  //  order to find the smallest
            repeat                          //  grouping value (GV) in Split
              ChildOp.Next()
              if Dflow➔GrpLst < Split.Value then
                Split←DFlow.GrpLst          //  Split converges to the
            until DFlow.Child = EOF         //  smallest GV at EOF
GBY.Next    if Split.Value ≠ +∞ then        // there is a next GV to compute
              Current←Split                 //  Initialize the computation
              DFlow.AggLst←0                //  of the current GV
              Split.Value←+∞                //  Initialize Split for computing
              ChildOp.Open()                //  the next GV
              repeat                        //  scan the whole input
                ChildOp.Next()
                if Dflow➔GrpLst = Current.Value then // the current tuple
                  compute DFlow➔AggLst      //  shares the GV
                elseif DFlow➔GrpLst∈] Current.Value,Split.Value [ then
                  Split←DFlow.GrpLst        //  Split converges
              until DFlow.Child = EOF
              DFlow.GrpLst←Current          //  Output an aggregate
Sort.Open   ChildOp.Open()                  //  Identical to GBY.Open
            Split.Value←+∞
            repeat
              ChildOp.Next()
              if DFlow➔SortLst<Split.Value then
                Split←DFlow.SortLst
            until DFlow.Child = EOF
            Current←Split
Sort.Next   ChildOp.Next()
            while DFlow➔SortLst ≠ Current.Value and
                  (DFlow.Child ≠ EOF or Split.Value ≠ +∞)
              if DFlow.Child = EOF then     //  The input ends but there is
                ChildOp.Open()              //  a sorting value (SV) in Split
                Current←Split
                Split.Value←+∞              //  Reinit. Split to find next SV
              elseif DFlow➔SortLst ∈ ]Current.Value, Split.Value[ then
                Split←DFlow➔SortLst         //  Converges to the next SV
              ChildOp.Next()
            // While loop ends when a tuple with Current SortLst is found
```

Figure 2: *RLB Algorithms*

3.4 GroupBy algorithm

The GroupBy operator aggregates in a single result tuple all input tuples sharing the same grouping value. By rule R2, *DFlow* contains a single tuple and then grouping values have to be processed one after the other. By rule R1, keeping track

of the list of all grouping values already processed by the algorithm is precluded. Thus R1 and R2 lead to process the grouping values in a predefined order, so that recording a single value sums up the history of the whole processing. The consequence is that the RAM consumption of the GroupBy algorithm amounts to two variables: *Current* referencing the grouping value being processed at each iteration and *Split* recording the frontier between the grouping values already processed and the ones remaining to be processed. This constitutes a RAM lower bound in the absence of assumption on the initial ordering of the input (hypothesis H1). Different RLB forms of the GroupBy algorithm can be devised depending on the way the *Split* variable is managed.

The first variation of the GroupBy algorithm, called *CompMin*, uses *Split* to reference the next grouping value to be computed. At the first iteration (implemented by Group.Open), the algorithm scans its input and searches into *Split* the smallest grouping value present in the input. At each next iteration, *Current* takes the value of *Split*, then the algorithm scans again its input and aggregates the tuples sharing the grouping value referenced by *Current*. At the same time, the algorithm searches into *Split* the grouping value to be processed at the next iteration, that is the value immediately greater than the one referenced by *Current*. This algorithm then performs (k+1) iterations on its input. Unicity and Completeness are guaranteed by the fact that the grouping values are processed in ascending order. Therefore, the same value cannot be considered in different iterations and all grouping values are considered. Indeed, Group.Open initiates the processing by finding the smallest grouping value and the stopping condition in Group.Next is that there is no grouping value greater than the last processed. In addition, each iteration scans the input sequentially, so that a tuple sharing the grouping value being processed is considered exactly once, assuming the child operator enforces Unicity and Completeness.

The second variation of the GroupBy algorithm, called *CompMax*, uses *Split* to reference the last grouping value that has been processed. During the first iteration, *Current* is used to converge to the smallest grouping value and to compute the resulting aggregate at the same time. To illustrate this, let assume the first grouping value encountered in the input be *v1*, so that *Current* references *v1*. While iterating on the input, tuples having a value higher than *v1* are not considered and tuples sharing the value *v1* participate in the aggregation. If a tuple having a value *v2* < *v1* is encountered, *Current* evolves to *v2* and the aggregation calculus is reinitialized. At the end of the first iteration, *Current* has converged to the smallest grouping value and the resulting aggregation has been computed. At each next iteration, *Split* takes the value of *Current* and *Current* is used to converge to the next grouping value (i.e., the value immediately greater than the one referenced by *Split*) and to compute the resulting group at the same time. While k iterations suffice to compute all aggregations, a $(k+1)^{th}$ iteration is required to guarantee the Completeness of the algorithm, that is to check that there exist no greater grouping value than the last processed. The remaining of the proof of Unicity and Completeness is equivalent to the CompMin algorithm.

Intuitively, and as its name indicates, *CompMax* does more job than *CompMin* since several aggregation calculus are partially performed before being discarded. However, combining the first iteration of *CompMax* with the next iterations of *CompMin* leads to a third algorithm, called *IterMin*, that implements the GroupBy in only k iterations. Indeed, the first iteration of *CompMax* does not use the *Split* variable. This variable could thus be exploited to determine the second grouping value to be processed, so that the (k-1) next iterations could be computed in the same way as *CompMin*. Figure 3 summarizes the behavior of each algorithm on an input containing a sequence G1<G2 …<Gk of grouping values. For each algorithm, we represent a snapshot of its internal state (i.e., Current, Split and Agg variables) at givens iterations. The gray arrow represents the current tuple being processed at a given iteration (e.g., if the tuple being considered at iteration 2 of CompMin is (G5,8), then the values of Current, Split and Agg are respectively G2, G3 and 0).

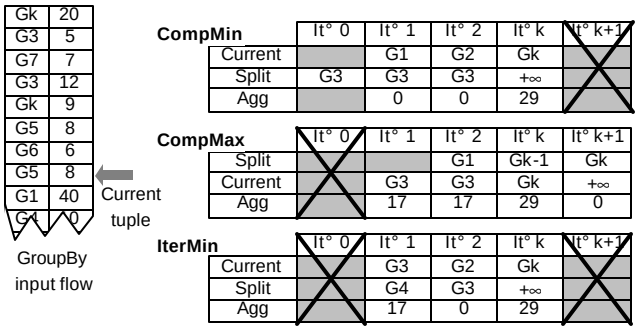

Figure 3: *Snapshot of the three GroupBy algorithms*

Sort algorithm

The Sort algorithm shares the same structure, RAM lower bound, number of iterations and proof of Unicity and Completeness as the *CompMin* variation of the GroupBy algorithm. The sole difference is that each iteration delivers the tuples sharing the sorting value referenced by *Current* instead of aggregating them. This algorithm is thus not discussed further in this section. Note that a CompMax-like variation of the Sort algorithm could be devised but it would lead to $|input|$ iterations (instead of k+1) to sort an input flow.

3.5 Concluding remarks

The RAM consumption of a whole QEP incurred by our execution model can be computed as follows. According to the queries of interest (see hypothesis H2), the sole operators that may be involved several times in the same QEP are Scan, Select and Join and their RAM consumption is zero. Therefore, the RAM consumption ascribed to the QEP's operators is the RAM consumed by the GroupBy and Sort operators, namely $2*|GrpLst| + 2*|SortLst|$, which corresponds to the size of their respective *Current* and *Split* variables. The size of the dataflow corresponds to the size of the *DFlow* structure plus the size of the current tuple being delivered in the final result, that is: $|FromLst| + |AggLst| + S_i size(a_i)$, for each $a_i \hat{I} p$, p being the Project condition.

Consequently, the RAM lower bound to execute a query without index can be expressed by:
$RLB = S_i\ size(a_i) + 2 \times |GrpLst| + 2 \times |SortLst| + |FromLst| + |AggLst|$
This result demonstrates the feasibility to design a query evaluator consuming a tiny bounded RAM, independent of the cardinality of the queried data and of the query complexity. One may doubt about the practical interest of RLB expecting that lightweight platforms will be equipped with a much larger RAM. Note however that current smartcard platforms provides not much than a few hundred bytes of RAM to the on-board applications, the rest of the RAM being preempted by the operating system, and the JavaCard VM. Regarding future platforms, smartcard manufacturers put more emphasis on the increase of CPU speed, communication bandwidth and storage capacity than on RAM. In addition, the objective of co-design is to lower the hardware resources (among them the RAM) to their minimum in order to save production costs on large-scale markets.

But beyond this formula, the significance of this study is on providing guidelines and algorithm's structures that remain valid when a small quantity of RAM is added to the architecture.

4 Optimizations in RAM lower bound

Not surprisingly, a RAM lower bound query execution model exhibits poor performance since every information that cannot be buffered in RAM needs to be recomputed. The dramatic consequence on the number of iterations performed on the queried data is illustrated in Figure 4. Assuming that k is the number of (R.a,S.b) distinct values present in the GroupBy input, the scan of T turns to be executed k*||R join S|| times.

Different and complementary solutions can be investigated to decrease this iteration cost without hurting the RAM lower bound. First, global optimization techniques can be used to rearrange the query tree in order to minimize the total number of iterations required to evaluate it. This can be done notably by pushing selections down to the QEP and by finding an optimal join ordering. In a RAM lower bound context, Left-deep trees outperform Right-deep and Bushy trees except for extreme values of the join selectivity's. In conducting our experiments, we observed that the Left-deep tree heuristic remains valid when a small quantity of RAM is added to the model. This confirms the intuition that right subtrees have to be minimized to decrease the cost of recomputing them for each tuple of the left subtree. The ordering of joins in the Left-deep tree depends on their selectivity and on the cardinality of the relations involved, as usual. The query execution in a RAM lower bound exhibits other interesting properties such as: (i) RAM consumption is independent of the number of Join and Select operators, (ii) Join and Select algorithms are order preserving and (iii) intermediate results are never materialized in the QEP. These properties allowed us to devise original optimization techniques detailed in [ABP03]. However, these techniques are not developed further in this paper because they have to be reconsidered when a small quantity of RAM is added to the model.

Second, local optimization techniques can be devised to decrease the number of tuples considered inside each iteration. Note that local optimization has here a different meaning than in the usual case since it applies to one iteration rather than to one operator. Having a deeper look on the algorithms presented in section 3 allows to split, at each iteration, the input flow of each operator into three distinct sets of tuples. As illustrated in Figure 4, *Relevant tuples* are tuples participating in the operator's result for the current iteration (e.g., tuples sharing the current grouping value being computed by a GroupBy). Obviously, this set of tuples cannot be reduced. *Required tuples* are tuples modifying the internal state of the algorithm without participating in the iteration's result (e.g., tuples modifying the *Split* variable of the GroupBy algorithm). These tuples are required to enforce the Unicity and/or Completeness of the algorithm and their number depends on the algorithm itself. Thus, the respective merit of different algorithms implementing the same operator can be compared with respect to the number of Required tuples they consider. Selecting the one minimizing this number is a good heuristic. Finally, *Irrelevant tuples* are all the tuples present in the input that are not *Relevant* nor *Required*. Eager pruning should take place to avoid producing Irrelevant tuples and carrying them in the QEP up to the operator. The optimizations related to Required and Irrelevant tuples are developed in the following sections.

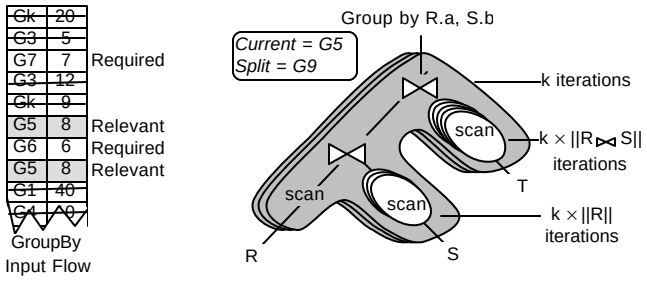

Figure 4: *Iterations performed by a RLB query evaluator*

4.1 Minimizing Required tuples

As stated earlier, the number of Required tuples considered by each algorithm depends on the way Unicity and Completeness are enforced. Regarding the Scan algorithm, one can notice that all tuples present in its input are Relevant. The inputs of the Select, Project and Join algorithms cannot contain Required tuples since they do not maintain an internal state. This comes from the fact that Unicity and Completeness are inherited from their child operator in the QEP. Therefore, these four algorithms are optimal in terms of Required tuples since this number is equal to zero.

The GroupBy algorithm is him directly impacted by the management of Required tuples. Let us first consider the *CompMin* variation of this algorithm. At a given iteration, all input tuples t such that t.GrpLst$\in$]Current, Split[are Required, where Current and Split denote the current state of the corresponding variables during this iteration. Indeed, each time an input tuple falls into this interval, it updates the Split variable thereby decreasing the upper bound of the interval. This is the way by which *CompMin* converges to the next grouping value to be computed and guarantees Unicity and Completeness. Thus, along the same iteration, the number of Relevant tuples (i.e., tuples sharing the grouping value

referenced by Current) remains constant while the number of Required tuples decreases and the number of Irrelevant tuples increases from the same amount, according to this converging process. In the *CompMax* variation of the GroupBy algorithm, the Required tuples at a given iteration are all the input tuples t such that t.GrpLst∈]Split, Current]. Each input tuple falling into this interval either lower the value of Current (if t.GrpLst<Current) or participate in the aggregation of the grouping value (if t.GrpLst=Current). This is the way by which *CompMax* converges to the grouping value to be computed at this iteration, computes it and guarantees Unicity and Completeness. Note that once convergence is achieved, all input tuples sharing the Current value are Relevant rather than Required. Let us now compare both algorithms with respect to the number of Required tuples they consider. Along iteration i (with i>1), *CompMin.Current* references the same value as *CompMax.Split* and, at the end of the iteration, *CompMin.Split* references the same value as *CompMax.Current*. However, the upper bound of the interval is open in CompMin while it is closed in *CompMax*. This strongly accelerates the convergence of this upper bound and makes *CompMin* much more efficient than *CompMax* in terms of Required tuples. Indeed, CompMin considers at each iteration at most one Required tuple per distinct grouping value remaining to be processed while *CompMax* considers at most all the tuples sharing these values.

Comparing *CompMin* to *IterMin* requires a deeper look at the first iterations. The first iteration of *IterMin* produces the same result as the first two iterations of *CompMin*, that is, detecting and processing the smallest grouping value. Again, converging to this value is faster in *CompMin* because at most one Required tuple has to be considered per grouping value. An algorithm considering less Required tuples cannot be envisioned without putting assumptions on the input ordering (hypothesis H1).

Thus, *CompMin* is preferred to *IterMin* and *CompMax* in the Ram Lower Bound context for it minimizes the number of Required tuples.

As the Sort algorithm shares the same structure as *CompMin*, it exhibits the same number of Required tuples. Thus, under hypothesis H1, the algorithms presented in Figure 2 are all optimal with respect to the number of Required tuples that need to be considered.

The number of Required tuples has an impact on the local cost of each algorithm. For example, all Required tuples participate in the computation of grouping values that turn to be discarded both in *CompMax* and in the first iteration of *IterMin*. But beside this local overhead, Required tuples have a much more negative impact on the global cost of the whole QEP. Indeed, they generate computations from the leaves of the QEP up to the operator that requires them, without participating in the iteration's result. Minimizing the number of Required tuples during an iteration amounts to maximize the number of Irrelevant tuples that could be pruned early in the QEP. The next section develops this point.

4.2 Eager pruning of Irrelevant tuples

The distinction between Relevant, Required and Irrelevant tuples depends on each operator. Once this distinction has been made, eager pruning of Irrelevant tuples can be implemented as follows. Each operator willing to eliminate the Irrelevant tuples from its input expresses a predicate, called *iteration filter*, that selects only the Relevant and Required tuples for a given iteration. This predicate will then be checked by the operators participating in the QEP subtree producing this input. Conceptually, an iteration filter is similar to a regular selection predicate that is pushed down to the QEP to eliminate the Irrelevant tuples as early as possible. However, iteration filters may involve several attributes issued from different base relations. So, they are more complex than regular selection predicates and care must be taken on the way they are checked to avoid redundant computations. In the following, we first describe how iteration filters are expressed, then we concentrate on the way they are checked.

4.3 Expressing iteration filters

The following discussion is conducted on an operator basis. The Scan and Project operators are not concerned by the expression of iteration filters since all tuples they consider are Relevant. Regarding the Select operator, expressing an iteration filter to eliminate the Irrelevant tuples present in its input turns to delegate the selection process to another operator belonging to the QEP subtree producing the Select input. This is nothing but pushing selections down to the QEP, as usual. The Join algorithm considers many Irrelevant tuples since, at each iteration i, the Relevant tuples from the right input are only whose matching with the current tuple t_i from the left input. Thus, a Join iteration filter is the instantiation of the join predicate with t_i. In a Left-deep QEP, a Join filter is unfortunately inoperative. Indeed, checking it in the right subtree leads to evaluate the join predicate twice per tuple. Note that Join iteration filters may keep a strong interest in a more general context.

GroupBy filters: in the *CompMin* variation of the GroupBy algorithm, a tuple *t* is Relevant if t.GrpLst= Current, while it is Required if t.GrpLst∈]Current, Split[, where Current and Split denote the state of the corresponding variables at the time the tuple t is processed. The GroupBy iteration filter, or GroupBy filter for short, is therefore a predicate of the form (t.GrpLst ≥ Current and t.GrpLst < Split). Note that the Split variable evolves during a same iteration. It takes the value +¥ at the beginning of the iteration and then converges to the value immediately greater than Current. This introduces a particular form of predicate that can be termed dynamic since its selectivity increases along a same iteration.

Sort filters: As already stated, the RAM lower bound version of the Sort and GroupBy algorithms share the same structure. Thus, the discussion on GroupBy filters applies as well to Sort filters and need not be repeated.

Checking iteration filters

The objective is to push the evaluation of iteration filters down to the QEP in order to prune Irrelevant tuples as early as possible. The place where an iteration filter can be checked depends on the relations it involves. Mono-relation iteration filters are simply checked by the corresponding Scan operator, at the leaf of the QEP. Multi-relation filters have to be decomposed into several predicates that are checked at different levels of the QEP. Let assume an iteration filter involving the relations R_1, R_2,… R_n appearing in this order in

the Left-Deep tree (that is, R_1 is the very left leaf of the QEP). This iteration filter is split into n independent predicates as follows. The first predicate applies to R_1 and is checked by the corresponding Scan. The i^{th} predicate applies to the result of the join between R_1, R_2,… R_i and is checked by the corresponding join operator, and so on up to the join with R_n. Figure 5 shows the instantiation of this mechanism for a multi-attribute GroupBy.

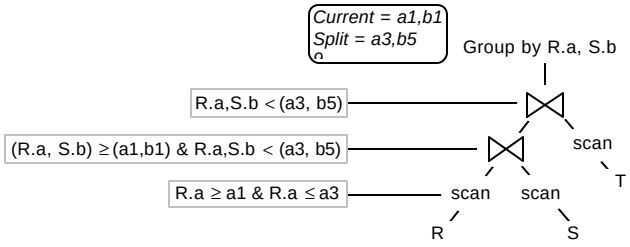

Figure 5: *Group Filters*

Let us have a closer look on this figure and consider a Required tuple t(x,y) produced by JoinRS. Before delivering it, JoinRS checks whether this tuple satisfies the GroupBy filter, namely t.GrpLst∈]Current, Split[(assuming the use of CompMin). Once delivered and considered by JoinRST, t can match with several tuples from T thereby producing a sequence of tuples of the form $\{tt_1, tt_2, \ldots tt_n\}$, from which only the first one is Required. Thus, tt1 will update the Split bound in such a way that $\{tt_2, \ldots tt_n\}$ become Irrelevant. The tuples $\{tt_2, \ldots tt_n\}$ have been produced vainly. To avoid such situation, the join algorithm has to be slightly modified in order to abort the processing of tuple t as soon as possible. This can be done by checking at each Next call from its parent that the current left tuple is still valid with respect to the iteration filter (i.e., t.GrpLst < Split).

While eager pruning of Irrelevant tuples is simplified by the Left-deep shape of the QEP, it can be extended to Right-deep and Bushy trees in a straightforward fashion.

5 The impact of RAM incremental growths

This section revisits the query evaluation and optimization techniques devised in a RAM lower bound context when a small quantity of RAM is added to the model.

5.1 Impact on the operator's algorithms

Let us first consider the impact of a RAM incremental growth on the design rules introduced in section 3.1. Rule R1 remains unchanged because variability plays against any memory bound. While RAM growth is considered, the RAM remains bounded by a small value. By incremental growth, we are expressing a slow deviation from the RLB bound, up to reach the value satisfying the application's constraints. Rule R2 could be reformulated as follows "recompute the information that cannot be stored in the bounded RAM". While this design rule seems obvious, it means that the philosophy of the algorithms remains unchanged, that is remains based on iterating – less frequently – on the operator's input(s).

Let us now consider the impact of additional RAM on each operator. The goal is to exploit RAM to materialize intermediate results, thereby reducing the number of iterations required on the operator's input(s). Scan, Select and Project operators are insensitive to the RAM size since they implement a single iteration on their respective input.

The Join algorithm benefits from additional RAM in a straightforward fashion. The nested-loop algorithm evolves to a block nested-loop, dividing the number of iterations on the right input by the number of buffer entries allocated to the join (assuming the buffered input be the left one). Tuple comparisons can also be saved by sorting or hashing the content of the buffer. In our context where the buffer is small, sorting is preferred to hashing since hashing consumes RAM on its own and would then decrease the useful part of the buffer.

The three variations of the GroupBy algorithm can benefit from a buffer. Let us first consider the CompMin algorithm. Exploiting the RAM available leads to divide the buffer into three arrays of a same cardinality *b*, called *Current[]*, *Split[]* and *Agg[]*. At each iteration (except the first one), the algorithm computes into Agg[] the *b* aggregations corresponding to the grouping values referenced by Current[] and searches into Split[] the *b* grouping values to be processed next. These next grouping values are determined thanks to a converging process, as in the RAM lower bound context. At a given iteration, all input tuples t such that t.GrpLst > max(Current[]) and t.GrpLst < max(Split[]) and t.GrpLst ∉ Split[] are Required. Indeed, each time an input tuple falls into this interval, it is inserted in sorted order into Split[] and the highest value of Split[] is discarded, thereby decreasing the upper bound of the interval. At each iteration, the algorithm considers at most one Required tuple per distinct grouping value remaining to be processed, as in the RAM lower bound context. However, since the number of iterations is divided by *b*, the total number of Required tuples considered by the algorithm is much less than in the RAM lower bound context. The CompMax algorithm can be buffered in the same way (i.e., by changing variables into arrays) except that a single Split variable is necessary to reference, at a given iteration, the highest grouping value previously computed. With buffering, the gap between CompMax and CompMin in terms of Required tuples increases since at each iteration, many aggregation calculus are partially performed before being discarded. On the other hand, CompMax produces less iterations than CompMin since more space is left to the Current[] array in the buffer. This makes the CompMax algorithm more efficient when iteration filters are not considered. The buffered extension of IterMin is not discussed since it has the same memory requirements as CompMin and considers more Required tuples.

The buffered adaptation of the Sort algorithm shares some similarities with CompMax. The buffer is divided into an array *Current[]* of b' entries dedicated to the storage of the smallest tuples to be delivered at a given iteration and *Split*, a single variable referencing the highest sorting value encountered at the previous iteration. At the first iteration, the algorithm scans its inputs and inserts the tuples in ascending order into Current[]. When Current[] overflows, the highest tuple in the sort order is discarded. At the end of this iteration, Current[] contains the b' smallest tuples corresponding to a sequence of sorted values of the form $v_1v_1v_1 < v_2v_2 < \ldots v_{n-1}v_{n-1}$

$<v_n v_n v_n$ (this sequence expresses the presence of duplicates wrt the Sort condition). All Current[] tuples having a sorting value in the range $[v_1, v_{n-1}]$ are then delivered in the sort order and Split is set to v_n (see Figure 6). At the next iteration, tuples of the input having a sorting value less than v_n are not considered, tuples sharing the value v_n are delivered and tuples having a sorting value greater than v_n are inserted in ascending order into Current[], and so on. At each iteration all input tuples t such that t.SortLst∈[Split, max(Current[])[can be either Relevant or Required and their status is actually determined a posteriori, at the time Split is reset (i.e., at iteration end). The Sort algorithm takes less benefit from a RAM growth than the GroupBy. This is due to the fact that tuples sharing the same sorting value have to be delivered together instead of being aggregated. Consequently, the performance of the Sort algorithm is driven by the number of duplicates wrt the sort condition.

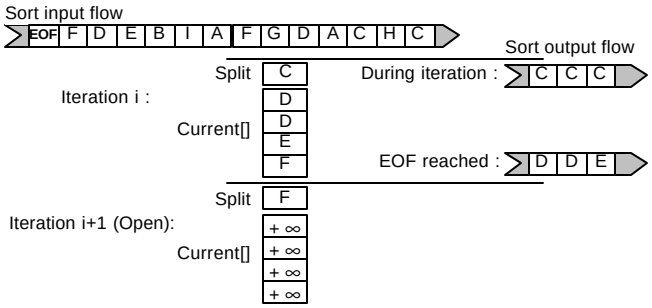

Figure 6: *Buffered Sort*

5.2 Impact on iteration filters

We focus below on the filter predicates expressed by the buffered version of the GroupBy and Sort algorithms. From the preceding section, it turns out that the predicate expressing a GroupBy filter for the CompMin algorithm takes the form (t.GrpLst∈Current[] or (t.GrpLst > max(Current[]) and t.GrpLst < max(Split[]) and t.GrpLst ∉ Split[])). We do not discuss the form of GroupBy filters for the CompMax algorithm since CompMin is always preferred when iteration filters are used. The predicate expressing a Sort filter takes the following form (t.SortLst∈[Split, Max(Current[])[). The predicates expressing these iteration filters are more costly to check than their RAM lower bound counterpart, although they are less frequently checked. Indeed, the inclusion of a tuple into an interval has now to be evaluated. Note that this evaluation can be accurate (exact match) or fuzzy (only the bounds are checked). The cost of an accurate evaluation is decreased by the fact that the GroupBy and Sort buffers are kept sorted.

Let us now consider the modification made on the join algorithm to check the filter predicates accurately (cf. section 4.3). Irrelevant tuples should now be discarded from a Join buffer as soon as the upper-bound of the filter predicate evolves. This leads to check all tuples present in the join buffer, on each Next call issued by the Join parent in the QEP. A less accurate alternative is to check only the current left tuple being considered. This simpler alternative has been shown more efficient by our performance evaluations.

6 Performance evaluation

The first objective of this evaluation is to assess the pertinence of the algorithms proposed in this paper, by quantifying the time required to execute different types of queries under strong RAM constraints. The expected outcome is to precisely evaluate to which extent these algorithms constitute an alternative to the use of indices. The second objective is co-design oriented. The expected outcome is here to provide valuable co-design guidelines by the means of curves helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and expected response time.

6.1 Experimentation platform

To conduct these experiments, we have implemented a complete query evaluator complying with the design principles introduced in this paper. The operator's algorithms are the ones presented in section 5.1. Regarding the GroupBy algorithm, CompMin is used when iteration filters are activated and CompMax is used otherwise. The QEPs of interest are optimized according to the heuristics described in this paper. Thus, they take the form of Left-deep trees where joins are ordered according to their respective selectivity and to the cardinality of the relations involved. The RAM is distributed on the resulting QEP thanks to a simple (and preliminary) cost model.

Counters are introduced in the platform to capture the number of elementary operations performed during the execution of a QEP (e.g., read a RAM cell, read a stable storage cell, evaluate a predicate …). These counters allow us to calibrate our prototype in order to reflect the behavior of different target hardware platforms (e.g., in terms of processor speed or stable storage technology). For this study, the platform has been calibrating with the following values: RAM read time = 50 ns, Stable storage read time = 100 ns, Processor speed = 50 Mips. These values correspond to advanced smartcard prototypes and are representative of the smartcard technology that will be available in the near future (two to four years).

6.2 Data, queries and experiments

In the scope of these experiments, we consider an on-board database composed of four base relations named *R*, *S*, *T* and *U*. Each relation contains at least three attributes: an integer primary key attribute, a string non-key attribute on which apply GroupBy and Sort operations and a string non-key attribute complementing the tuple to reach an average size of 100 bytes. In addition, foreign key attributes are added into *S* to reference *R* and into *T* and *U* to reference S. These relations are populated by a pseudo-random generator allowing us to generate either a uniform or a Zipfian (i.e., skewed) distribution of the data. The tuple cardinality of each relation, for a scale factor SF=1, is: $|R| = 100$, $|S| = 300$, $|T| = 1200$ and $|U| = 600$, leading to a 220KB database.

We consider different types of queries, summarized in Table 1, of increasing complexity. Queries Q1 to Q5 are named *Regular* for they are representative of usual queries that we could foresee in an embedded context (sensors, personal folders, ambient intelligence). The simplest Regular

query (Q1) computes a single join while the most complex ones (Q4 and Q5) compute two joins followed by a GroupBy or a Sort. To make the study complete, we consider also *Complex* queries (Q6 and Q7) involving three joins and a multi-attribute GroupBy or Sort.

6.3 Interpretation of the results

Let us first study how the query evaluator behaves in the presence of Regular queries. The curves plotted in Figures 7(a) to 7(e) represent the respective execution time of queries Q1 to Q5 as a function of the RAM. On each figure, the plain curve represents the execution time required to execute the query without iteration filters while the bold curve integrates the benefit provided by iteration filters. These two curves divide the space into three areas. The area above the plain curve materializes all combinations of response time (RT) and available RAM that can be reached by exploiting our operator's algorithms, without iteration filters nor index. The area delimited by the two curves represents the RT/RAM combinations that become accessible by adding iteration filters to the preceding algorithms. Finally, the gray area represents the RT/RAM combinations that are unreachable without index. A fourth area on the left end side of each figure shows the combinations that can never be considered since they are located under the RAM lower bound. These curves deserve two important remarks. First, the hyperbolic shape of the curves shows that the operator's algorithms exploit very well any RAM incremental growth. To illustrate this, the time required to execute Q2, without index nor iteration filter, amounts to 75 seconds with 150 bytes of RAM and falls down to 12 seconds with an addition of only 100 bytes of RAM. Second, the iteration filters strongly enlarge the area where the resort to indices can be avoided. Typically, the quantity of RAM required to execute Q3 in 1 second without iteration filters is 2,5KB and falls down to 1,2KB when iteration filters are exploited. For queries Q4 and Q5, the benefit of iteration filters seems graphically less impressive but this feeling is only due to the logarithmic scale of the figure. For example, executing Q5 in 1 second requires more than 16KB of RAM while 2KB suffice when iteration filters are used.

The main conclusion of this first series of experiments is that the combination of our operator's algorithms with iteration filters constitutes a very convincing alternative to the use of indices for the considered queries. Note that, thanks to these techniques, Q1 to Q5 can all be executed with a response time close to 1 second (the worst case being 1,4 second for Q5) with only 1KB of RAM. This result is particularly significant considering that the domain of investigation delimited by a response time around 1 second and a RAM around 1KB seems to be very challenging. Indeed, 1 second represents a « psychological » barrier reflecting well the requirement of most interactive applications. 1KB of RAM could however be considered as a two extreme bound and one may wonder whether not to consider more comfortable assumptions regarding the RAM resource. The first reason is technological. Today's ultra-light devices like smartcards are equipped with 1 to 4 KB of RAM (for the most powerful ones) but only a few hundred of bytes is left to the on-board applications, the rest being consumed by the OS, the JVM and the execution stack. Thus, 1KB of RAM allocated for query processing is today a rather optimistic assumption and semiconductor manufacturers do not forecast a rapid growth of the RAM resource for several reasons like reducing the silicium die size, the power consumption and the security threats. The second reason is economic and leads to lower all hardware resources (among them the RAM) to their minimum in order to save production costs on large-scale markets. So if more RAM is not expressly required, it will not be provided.

Figures 7(f) and 7(g) gives another insight into the gain brought by iteration filters. Figure 7(f) expresses the ratio between the execution times obtained without and with iteration filters for the two queries Q2 and Q3 as a function of the RAM. Not surprisingly, the lower the RAM the higher the ratio since the RAM determines the number of iterations performed by the GroupBy and Sort algorithms. Note that this ratio would be even greater with queries Q4 and Q5 since the GroupBy and Sort algorithms would reiterate on a bigger subtree. The highest benefit is measured in the range [RLB, 1,5KB] of RAM. This motivates further the use of iteration filters when the RAM resource of a device has to be minimized. Figure 7(g) plots the percentage of RAM saved in the execution of Q2 and Q3 by iteration filters as a function of the expected execution time. For example, the amount of RAM required to execute Q2 in less than 1 second is 200 bytes with iteration filters and 940 bytes without iteration filters, leading to a gain of 78% of RAM.

Figures 7(h) and 7(i) evaluate the robustness of our algorithms against an increase of distinct values and skewed data. The evaluation of Q2 and Q3 is measured with and without iteration filters for a given amount of RAM of 1KB. As shown by Figure 7(h), the use of iteration filters makes the operator's algorithm more stable when facing more distinct values. This phenomenon is due to the eager elimination of the Irrelevant tuples that participate in an increasing number of iterations. Figure 7(i) shows that the GroupBy algorithm is insensitive to skewed data, the number of iterations remaining constant. The Sort algorithm takes advantage of skewed data because several sorting values shared by few tuples can be processed at the same iteration.

Finally, the last four figures evaluate to which extent our algorithms scale when the query complexity or the volume of data augments. Figure 7(j) illustrates the benefit of iteration filters for a complex query involving 3 joins and a multi-attribute GroupBy. For 1KB of RAM, iteration filters reduce the cost of the execution from 9,6 to 1,4 seconds. Clearly, the more complex the query, the more efficient the iteration filters. Figure 7(k) plots the execution time of the filtered execution of all queries of interest as a function of the RAM. This figure demonstrates that the proposed algorithms scale well when they face complex queries. Indeed, all queries except Q7 can be executed around one second (the worst case being 1,4 second for Q5 and Q6) with only 1KB of RAM. Figure 7(l) plots the execution time of the filtered execution of all queries as a function of the database size for a fixed quantity of RAM of 1KB . The first learning of this figure is that our algorithms scale pretty well for Regular queries. However, they scale badly in the presence of Sort or in the

Query	Query Type	Output Tuples / Dist. Values
Q1	Join(S, T)	1200
Q2	GroupBy(S.a, join(S, T))	60 / 60
Q3	Sort(S.a, join(S, T))	1200 / 60
Q4	GroupBy(R.a, Join(R, S, T))	1200 / 20
Q5	Sort(R.a, join(R, S, T))	1200 / 20
Q6	GroupBy(R.a, S.b, join(R, S, T, U))	200 / 200
Q7	Sort(R.a, S.b, join(R, S, T, U))	2400 / 200

Table 1: *Queries Description*

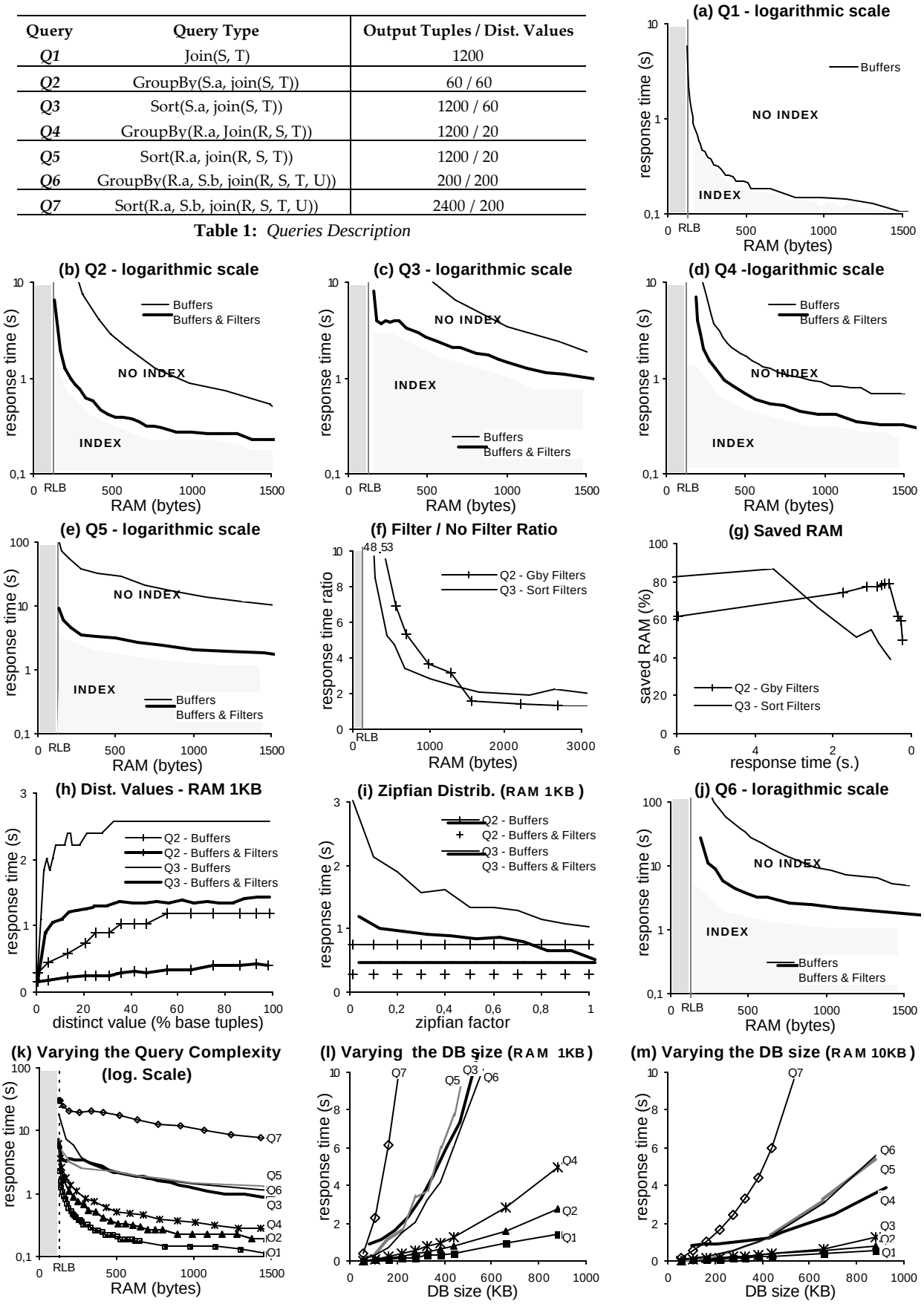

Figure 7: *Evaluation results*

presence of several joins. This situation was predictable and one cannot expect execute complex queries on a large database without index and with only 1 KB of RAM in less than 1 second. Two ways can be investigated to decrease the query execution time, namely adding more RAM or adding indices. Figure 7(m) plots the same curves with a larger RAM of 10KB and shows that the problem becomes less critical without totally disappearing and that complex queries involving Sort are still not tackled. Adding indices is the way followed by PicoDBMS [PBV01]. It solves the performance problem at the price of the side effects mentioned in section 2.

As a conclusion, these experiments show the accuracy of the proposed operator's algorithms along with their iteration filters and demonstrate that they constitute a real alternative to the index in a wide range of situations. Beyond this range, indices should be considered. Finally, note that all the curves presented in this section can be used for co-design purpose. Indeed, they provide valuable information to determine: whether indices or iteration filters are required in a given situation, how much RAM should be added to reach a given response time, how much the expected response time should be relaxed to tackle a given query with a given quantity of RAM and how much data can be embedded in a given device without hurting an expected execution time.

7 Conclusion

Pervasive computing and ambient intelligence motivate the development of new data-centric applications that must be tackled in a growing variety of ultra-light computing devices. As far as query execution is concerned, RAM appears to be the most critical resource in these devices. In the absence of a precise understanding of the RAM consumption problem, ad-hoc solutions have been developed. Most of them introduce strong restrictions on the type of queries and on the amount of data that can be tackled while the others resort to dedicated index methods having negative side effects. This introduces the need for pre-designed database components that can be integrated in Systems on Chip.

This paper precisely addresses this issue and proposes a framework helping to design RAM-constrained query evaluators. First, we proposed a query execution model that reaches a lower bound in terms of RAM consumption. Second, we devised a new form of optimization, called iteration filter, that drastically reduces the prohibitive cost incurred by the preceding model, without hurting the RAM lower bound. Third, we proposed variations of the preceding techniques that best exploit any incremental growth of RAM. Our performance evaluations led to two important and practical outcomes. First, they show the accuracy of the proposed techniques and demonstrate that they constitute a convincing alternative to the index in a wide range of situations. Second, they provide helpful guidelines helping to calibrate the RAM resource of a hardware platform according to given application's requirements as well as to adapt an application to an existing hardware platform.

While this paper draws the limit beyond which indices are required, an interesting future work is to study the combination of our operator's algorithms, iteration filters and indices. Our feeling is that these solutions can fit well together and can cover a very large range of situations in the most accurate way. Another important issue is to put these results in practice. A cooperation has been set up with Schlumberger to study the evolution of their smartcard operating system to tackle on-board data centric applications.

8 References

[ABB01] N. Anciaux, C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, "PicoDBMS: Validation and Experience", *Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[ABP03] N. Anciaux, L. Bouganim, P. Pucheral, "On Finding a Memory Lower Bound for Query Evaluation in Lightweight Devices", Technical Report, PRiSM *www.prism.uvsq.fr/rapports/2003/document_2003_40.pdf*

[BGS00] P. Bonnet, J. Gehrke, P. Seshadri, "Querying the Physical World", *IEEE Personal Communications Special Issue on Networking the Physical World*, 2000.

[CaK97] M. J. Carey, D. Kossmann, "On Saying "Enough Already!" in SQL", *Int. Conf. on Management of Data (SIGMOD)*, 1997.

[GDM98] R. Gupta, S. Dey, P. Marwedel, "Embedded System Design and Validation: Building Systems from IC cores to Chips", *Int. Conf on VLSI Design*, 1998.

[GIG01] E. Giguère, "Mobile Data Management: Challenges of Wireless and Offline Data Access", *Int. Conf. on Data Engineering (ICDE)*, 2001.

[Gra93] G. Graefe. "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, 25(2), 1993.

[Gra98] G. Graefe. "The New Database Imperatives", *Int. Conf. on Data Engineering (ICDE)*, 1998.

[ImN02] T. Imielinski, B. Nath, "Wireless Graffiti – Data, data everywhere", *Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[ISO99] Int. Standardization Organization (ISO), Integrated Circuit(s) Cards with Contacts – Part 7: Interindustry Commands for Structured Card Query Language-SCQL, ISO/IEC 7816-7, 1999.

[KLL01] J. S. Karlsson, A. Lal, C. Leung, T. Pham, "IBM DB2 Everyplace: A Small Footprint Relational Database System", *Int. Conf. on Data Engineering (ICDE)*, 2001.

[MFH02] S. Madden, M. J. Franklin, J. Hellerstein, W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", *Int. Conf on Operating Systems Design and Implementation*, 2002.

[MH02] S. Madden, J. M. Hellerstein, "Distributing Queries over Low-Power Wireless Sensor Networks" , *Int. Conf. on Management of Data (SIGMOD)*, 2002

[NRC01] NRC report, *Embedded Everywhere, A research agenda for networked systems of embedded computers*, national academy press, 2001.

[Ora02] Oracle Corporation, *Oracle 9i Lite - Oracle Lite SQL Reference*", Oracle Documentation, 2002.

[PBV01] P. Pucheral, L. Bouganim, P. Valduriez, C. Bobineau, "PicoDBMS: Scaling down Database Techniques for the Smartcard", Very Large Data Bases Journal , 10(2-3), 2001.

[SeG01] P. Seshadri, P. Garrett: "SQLServer for Windows CE - A Database Engine for Mobile and Embedded Platforms", *Int. Conf. on Data Engineering (ICDE)*, 2000.

[Ses99] P. Seshadri, "Honey, I Shrunk the DBMS': Footprint, Mobility, and Beyond", Int. Conf. on Management of Data (SIGMOD), 1999

[SIA02] Semiconductors Industrial Association, "STATS: SIA Annual Databook", 2002. http://www.semichips.org