Computing Partitions within SQL Queries: A Dead End?

Frédéric Dumonceaux – Guillaume Raschia – Marc Gelgon

December 20, 2012

Abstract

The primary goal of relational databases is to provide efficient query processing on sets of tuples and thereafter, query evaluation and optimization strategies are a key issue in database implementation. Producing universally fast execution plans remains a challenging task since the underlying relational model has a significant impact on algebraic definition of the operators, thereby on their implementation in terms of space and time complexity. At least, it should prevent a quadratic behavior in order to consider scaling-up towards the processing of large datasets.

The main purpose of this paper is to show that there is no trivial relational modeling for managing collections of partitions (*i.e.* sets of sets). In the withheld case, we show that one could not express all the operators of the partition lattice and set-theoretic operations of the algebra of sets (viewing blocks as elements) within FO, and consequently as queries of the relational algebra (RA). We also show multiple evidence of inefficiency of RA-expressible operators and an alternative which warrant another computational model. Further, we present some experimental results that enforce this evidence and conclude that R-DBMS are inadequate for partition querying. Hence, we claim that there is a strong requirement for the design of an *ad hoc* system to manage partitions or at least to supplement an existing system on which both data persistence and transaction management could be delegated.

1 Introduction

Partitions are one of the most popular data structures.

Much more than list, queue, tree and record, partition is basically the output structure that has been widely adopted by practitioners in many fields and application domains, as much as graphs for network modelling. Indeed, partitions are results of cluster analysis, image segmentation, spatial decomposition, segmentation in distributed systems and parallel computing, etc. Partitions are also the preferred underlying structure of many basic views over the data such as pies, charts, histograms, statistical data on maps[1] and it may be relevant to store those many views for reuse and mashup in the future, rather than redrawing at query time the underlying partition structure.

Furthermore, partitioning a data set into groups is one of the central tasks of data mining. One main reason is that partitions reveal some aspects of the structure of the underlying process (physical, social, ...), by separating subpopulations. As an example, graph clustering techniques for community extraction is a very popular application. Using partitioning algorithms may follow a simple workflow: a) an algorithm is executed on a given data set, b) it supplies one partition and c) it lets a human expert or end-user exploit the result.

Further, data mining tasks may themselves consider multiple data partitions as their input, besides data sets. In particular, the scheme coined as *ensemble clustering*, recently discussed in [14], aims at building a partition that better reveals the structure in the data than individual partitions, by combining these individual partitions. Implementing such techniques often requires operations, elementary or not-so-elementary, on partitions. Such a task is often cast as the *median partition* optimization problem for which the optimal solution is then the consensus agreed by all partitions and yield by computing several *meet* operator between each operand, namely the greatest lower set of relations between pair of entities clustered.

Finally, there is currently a strong shift towards moving data mining, visualization and interpretation from the hand-cared small-size lab experiment to on-line and collaborative data repositories and services. Repositories may store a large number of data sets, each of which may be associated to several partitions describing it. In particular, large-scale scientific analysis over shared data sets [12] gains from effective shared infrastructures to cooperate or compete, and capitalize as much as possible on each others' results. This setting is likely to produce many partitions over a shared data set, produced by several users. It may occur that several data sets on a repository share individuals and/or variables, or otherwise relate to each other. For instance, in physics, medicine, or social sciences, as a new variable is made observable, there is interest in assessing how this affects the way individuals are clustered.

[1] See for instance Hans Rosling talk at TED'2006.

Problem

Through these cases, we tried to demonstrate that there are strong needs for data management systems to support manipulations on data partitions with a dedicated well-founded modeling framework. In particular, the aim of our paper is not to find a way to address the issue to represent one particular partition or a subset; we aim to deal deliberately with all partitions of a set and that involves a study of the most likely abstract computation model to fit both algebraic *w.r.t.* the whole and computation properties. To prevent from reinventing the wheel, a first attempt would be to match set partitions with the well-established Codd's *relational model* [5], that provides mature systems with sophisticated query capabilities, query optimization strategies and well-founded theoretic background.

Related Work

Many complex data structures require some special forms of representation. For instance, temporal data, spatial data, multimedia data, objects, semi-structured and unstructured data, documents, etc. None of these data types fit well within tables. Among the many native data models for storage systems that have been studied in the last decade, it comes out that complex objects and trees and graphs (and documents) are too powerful to efficiently compute set partition queries and key/value is basically too restrictive.

Some ongoing research were carried out for the further development of efficient modeling of complex data structures by means of the relational encoding. Although the relational model was early pointed out for its difficulty to encode some several complex structures, *relational query processors* supported by the *well-founded relational algebra* are undoubtedly the most studied query engines and have gained a great popularity from scientists and practitioners working around the topic of database systems.

A straightforward extension yields to N1NF-relations [1] and their nested relational algebra counter-part. Also they provide a basic mean to encode partitions as sets of sets, physical models and query engines mainly rely on flat relation-based optimizations such that they do not offer much more efficiency than regular relational encoding.

For instance, some efforts have been done to support an algebra of sets in SQL through set-comparison queries involving several nested queries since each set-theoretic operator requires its semantical translation in predicate calculus restricted to existential quantifiers [10].

Further, lots of works, such as [13, 3, 2], have been conducted to provide with XML-relational mapping and object-relational mapping in many directions.

Although the objective is similar in the sense that they try to map complex structures into relations, none of the above approaches deal with set partitions.

Orthogonally, there are several works [15, 11, 7], most of which deal with theoretical aspects, that handle set partitions and their operations. Halverson et al. [11] give a very formal overview of set partitions and their algebraic prop-

erties. In [15], authors define a Minimum Description Length problem for set partition-like data structures and they apply to query optimization purpose for ROLAP queries. Another effective approach to set partitions is the so-called *relation partition algebra* (RPA) proposed in [7]. In the application domain of software engineering, RPA brings mathematical foundations for modular development b.t.w. of the lifting operator defined as a binary relation ("use" and "part-of" module dependencies) over equivalence classes of a partition i.e., all functions packaged within the same module. This work focuses on the relational level built on top of the partition algebra, rather than studying partition algebra itself.

As far as we know there has been no assessment of how does computation on set partitions perform when the data model is following a relational-based encoding whatever would be its flavor.

Contribution

We then propose to study the *partition-relational mapping* and draw some conclusions about the "impedance mismatch" problem with partitions. The overall contribution of the paper is as follows:

- Discussion and design of a relational encoding scheme for partitions through membership-based approach;
- Relational queries for the main partition operators against a dedicated encoding scheme;
- Optimized operators and their implementation in SQL queries.
- Performance evaluation of SQL is carried out by experiments using realistic generator for partitions and according to a C++ implementation.

Overview

The following of the paper is organized as follows. Section 2 presents the basic mathematical background required to operate on set partitions in different flavors. Section 3 introduces the encoding schemes for partitions, based on their mathematical definition. In Section 4, we give relational queries against encoding schemes that relate to a collection of set partition operations. Finally, Section 5 is dedicated to experiments and provide with performance values that enforce our static analysis.

2 Data Model(s)

It comes as no surprise that there is a very close relationship between the concept of *basis* in abstract algebra and the choice of an explicit representation for a large variety of types of data. Any such basis should offer to express any atomic object independantly of a specific representation space, i.e. geometric realization, in

order to consider formal interactions between objects in a combinatorial fashion. The choice of a *canonical* basis is an unavoidable task to ensure both a unique representation of each object mathematical and a clear representation that is by an appropriate combination of self-contained objects, in order to span every other objects.

It is therefore obvious that a partition of a set disclose two level of nesting and thus is endowed with three logical layer of abstraction. The support set is basically a set of entities under which a part or all of these is in relationship with a particular subset to highlight ongoing relations. In the purpose where it is not relevant to quantify the established relathionship between two entities, this clearly embodies a relaxation over constraints to ensure consistency of relations between entities in the scope of the cluster. The last layer is merely topped by the partition itself and taken as a whole, i.e. where any partition is viewed as a first-class object within the data management system.

Firstly, We shall review some useful facts about partitions of a set and its underlying algebraic structure.

2.1 Partition Algebra

Let Ω be a finite and countable set of objects; let a_1, a_2, ..., a_n be subsets of Ω. Then $P = \{a_1, a_2, \ldots, a_n\}$ is a partition of Ω, if and only if it is a set cover of Ω, and each block is not overlapping with each other. For ease of reading, we shall not use the set-standard writing of partitions and denote $P = a_1|a_2|\ldots|a_n$ where $|$ seperate blocks. Thereafter, we use natural numbers $\mathbb{N}$ as the underlying domain for Ω without loss of generality.

The set of all partitions Π_Ω defined on the same ground set Ω is usually endowed with the *refinement relation* $\preceq$ so that the partially ordered set $(\Pi_\Omega, \preceq)$ is the well-known *partition lattice* where Q *refines* P, denoted $P \preceq Q$, if and only if every block of Q is a (strict or not) subset of a block in P. We also define the down set of a partition P by $\downarrow \{P\} := \{R \mid P \preceq R\}$, generating a sub-lattice under inclusion.

According to the algebraic definition of the partition lattice $(\Pi_\Omega, \Cap, \Cup)$, there are semantically equivalent definitions for $\Cap$ and $\Cup$ operators by means of their *least upper bound* and *greatest lower bound*:

$$P \Cup Q := \min \sup\{R \mid R \preceq P \wedge R \preceq Q\}$$
$$P \Cap Q := \max \inf\{R \mid P \preceq R \wedge Q \preceq R\}$$

and leads to $(P \preceq Q) \iff (P \Cup Q = P) \wedge (P \Cap Q = Q)$ and $\top_\Omega \preceq P$ and $P \preceq \bot_\Omega$ for all $P, Q \in \Pi_\Omega$. Moreover, since the partition lattice is atomic, there is a minimal decomposition (non-unique) of a partition as a join ($\Cup$) of partitions called *atoms* (themselves self-contained) and denoted as $J(\Pi_\Omega)$, which directly covers the singletons partition so that $P \prec \bot_\Omega$, $\forall P \in J(\Pi_\Omega)$. The rank of any partition P in the partitition lattice is defined by $\mathrm{rk}(P) = |\Omega| - |P|$.

Besides, partitions can be viewed are "sets of sets", we also would like to consider some set-theoretic operators that apply on blocks. Indeed, set partitions might be viewed as subsets of the *powerset algebra* 2^N where $N \leq |\Omega|$ and

$\forall P \in \Pi_\Omega, P \in 2^{2^\Omega}$, thereby we can apply some boolean operations on pairs of blocks.

$$P \cap Q := \{a \mid a \in P \wedge a \in Q\}$$
$$P \cup Q := \{a \mid a \in P \vee a \in Q\}$$
$$P - Q := \{a \mid a \in P \wedge a \notin Q\}$$

As usual, intersection operator is equivalent to computing either $P - (P - Q)$ or $Q - (Q - P)$. Both $-$ and $\cap$ are well-defined over partitions, given that they build partitions on support sets $\mathcal{S} \subseteq \Omega$. Indeed, only a subset of blocks from P composes the result set of $P \cap Q$ and $P - Q$ as well. It is worth to notice that union operator is *unsafe* since raw union of partitions can output overlapping blocks, then union operation $P \cup Q$ is consistent if and only if $P = Q$ and hence, it is not considered as a valid operation in Π_Ω.

Furthermore, applying some of these operators is similar to project one partition onto another one in the lattice structure such as repeated this one leads to the same, then such operators are called *idempotent* since $P \ \mathsf{Op} \ P = P$ and $\mathsf{Op} \in \{\Cup, \Cap, \cap\}$ hold, then $\top, \bot$ are trivial idempotent partitions whereas P, Q are mutually orthogonal if and only if $P \Cup Q = \top_\Omega, P \Cap Q = \bot_\Omega$. In a RI approach, this revealed that pair of orthogonal partitions provides a general summary on how pointless their underlying relations are involved in a consistent whole, *i.e.* over others instances of a collection, since any combination reach either $\top_\Omega$ or $\bot_\Omega$ and hence is not consistent to some extent to the end-user.

2.2 Extensional representation

Design of a relational representation of a set partition must be achieved *w.r.t.* some prerequisites. In the algebraic sense, such a representation could be seen as a collection of distinct objects, when combined to form various partitions should express explicitly the useful knowledge about each object assignement to their respective block. A well-designed representation should permit to, at some extent, easily query what is the block of any object in the partition scope, that is, apply set operators $\{\cap, -\}$ which involves to run boolean tests on the whole blocks.

In a similar way, lattice operator $\{\Cup, \Cap\}$ requires the knowledge about both clusters and objects, and implies that a property should be preserved to enclose objects into the former block without doing explicit nest and unnest to switch between each scope. According to the semantic of the lattice operator, doing a query on any entry, means that we need to foresee solutions in a combinatorial approach during the calculation since every object viewed as distinct stored-resource, can be exploited more than once during the computation. If we care about ($\Cup$) computing on a couple of partitions, a somewhat naive method is to greedily merge blocks by pairs (each taken in distinct operand) whether they overlap. We would be able to take care that computing all explicit pairings does not ensure that each block shall be joining once in the relevant outcoming block.

Partition decomposition

Let the classical set-theoretic representation of partition be denote as the *intension* and conversely, an *extension* refers to an equivalent representation and ensures that same properties are disclosed and there exists an one-to-one mapping ε which conveys the same structure between each representation.

A trivial extension of a set partition relies on the *setoid* alternative representation, that is $(\Omega, \sim_\Omega)$ where $\sim_\Omega$ is an equivalence relation on the ground set Ω (reflexive, symmetric and transitive). Indeed, the partition lattice $(\Pi_\Omega, \preceq)$ is isomorphic to the lattice of the equivalence relation over Ω ordered by inclusion $(\sim_\Omega, \subseteq)$. This extension simply emphasizes that, given two entities $x, y \in \Omega$, x and y are in the same block in a partition P is denoted $x \sim_P y$.

Let us define the morphism $\phi : P \mapsto \sim_P$ as:

$$\phi(P) := \bigcup_{(a,a) \in P \times P} a \times a$$

and retrieve genuine partition is computed by its inverse such that $\phi^{-1}(\sim_P) := P/\sim_\Omega$ and hence recover explicit object assignement to their respective block.

Atoms are then an algebraic basis for any partition from the partition lattice since $\phi(P)$ is exactly the intersection between $J(\Pi_\Omega)$ and $\downarrow \{P\}$ (modulo both reflexive and symmetric relations). This representation is thus the most expensive description according to its space complexity which is in $O(|\Omega|^2)$. We come up with the following equalities:

$$\begin{aligned} \phi(P \Cap Q) &= \sim_P \cap \sim_Q \\ \phi(P \Cup Q) &= \langle \sim_P \cup \sim_Q \rangle \end{aligned}$$

Indeed, we need an algebraic closure operator $\langle \mathcal{R} \rangle = \bigcap \{ \sim \subseteq \Omega \times \Omega \mid \mathcal{R} \subseteq \sim \}$ in order to preserve $\sim_P \cup \sim_Q$ as be a closed set under union operatition. We have to check that Π_Ω is a semimodular lattice so that $\mathrm{rk}(P) + \mathrm{rk}(Q) \geq \mathrm{rk}(P \Cup Q) + \mathrm{rk}(P \Cap Q)$. For instance, let us say $|\Omega| = 3$ no matter what it contains, $\top_\Omega$ being the only partition expressible in terms of atoms, of which there are $\binom{3}{2} = 3$ in $J(\Pi_\Omega)$. It comes that the top partition ($\top$ is used here for convenience only) is expressible as the ($\Cup$) of two atoms in three different ways and $j_1 \Cup j_2 \Cup j_3 = \top \iff \sim_{j_1} \cup \sim_{j_2} \cup \sim_{j_3} = \sim_\top$ holds and then $j_1 \Cup j_2 = \top \iff \sim_{j_1} \cup \sim_{j_2} \subset \sim_\top$ also, according to atoms definition.

While each atom is distinguished from the others, they still shared objects from Ω and that leads to compute the transitive closure whose result includes one or several atoms. Override this closure is trivially ensured if and only if joining two partitions involves union of their respective atoms set does not include any pair of atoms which overlap and hence atoms set of the resulting partition forms itself a partition of Ω. To sum up, this condition endorses that the closure is no longer necessary and that every partition can have a single decomposition.

Moreover, for any partition P thus defined, its atoms set is itself a partition over a proper subset of Ω and yields to consider them as a set of independant

objects, let say a basis, whose composition (by union operation) is a boolean algebra $2^{\Omega'}$. It comes naturally that it does not match every partition defines over Ω whereas $|\Omega'| \leq \left\lfloor \frac{|\Omega|}{2} \right\rfloor$. In addition, let us define a second basis in order to reach a wider subset of Π_Ω such that $|\Omega''| = \left\lfloor \frac{|\Omega|}{2} \right\rfloor - |\Omega'|$, Ω to be entirely covered by any atoms included in Ω', then, each atom in Ω'' shall overlap with at least one in Ω' and we can't prevent us to compute a transitive closure.

Finally, trying to express a complete basis as a family of finite generating sets of disjoint strict subsets of Ω for a partition lattice bring back to an equivalence relation and it yields to compute a transitive closure.

The key issue is thus to avoid such an effective calculus in such a way to prevent quadratic storage space complexity and retrieval of explicit object membership when applying set operators $\{\cap, -\}$ which requires to compute:

$$\phi(P \ \mathsf{Op} \ Q) = \phi(\phi^{-1}(\sim_P) \ \mathsf{Op} \ \phi^{-1}(\sim_Q)), \quad \mathsf{Op} \ \in \{-, \cap\}.$$

Tree-based representation

Thankfully, since we are dealing with crisp relations enclosing objects within a block, such that they are indistinguishable one with each other, the transitivity property can be relaxed under this assumption as a *reachability* property. From a graph-theorical perspective, we shall consider as consistent the fact that x, y, z are equivalent *w.r.t.* a partition if there is a directed path ensuring that is a strong connected component, rather than a complete (sub)graph.

Such a structure is optimal according to its space complexity if it is a minimum spanning tree, *i.e.* $O(n)$-space, which entails every object with at least two edges (except for the leaves) and highlight that all objects within a block is reachable by all other ones. Two singular tree-based structures clearly stand out:

- a broken circuit, *i.e.* a chain graph, where all objects come upon it bumper-to-bumper, one among isolated objects is choosen as a root;
- a 1-level tree, *i.e.* a star graph, where one arbitrarily object is choosen among the block to be the root.

The chain case is not relevant and should be disregarded since a lookup is in linear time in the worst case while the star case warrants an access in $O(1)$-time. Furthermore, it can be easily traversed through a relational encoding since it is utmostly flattened.

Otherwise, we can rely on a generic tree-based representation and pre/post traversal based encoding [9], used to store XML-trees in SQL Table through a set of index each encoding a boolean algebra for any partition instance, likely interpreted as a singular basis, and discloses each alternative of coarsening, *i.e.* inverse relationship of refinement. In the other hand, it should be noted that there is a large number of possible paths since number of generated partitions grows exponentially with cardinality of the basis and leads, in default thereof,

to duplicate any partition as many times as there are paths to reach it and that is definitively an issue *w.r.t.* search space. Moreover, in order to rely partitions taken in different index, we should use an explicit representation for any element of the basis describing each instance since, for example, roots only provide a summary for the underlying block and those ones do not identify the same in two different indexes and even then, no exact matching can be done since in lack of a transitive closure, no partitions shall be caught in two different indexes. Design consistent index(es) for the partition lattice suggests in itself, further research beyond the scope of the present paper.

The withheld extension afterwards is the tree-based representation with 1-level where the root is choosen deterministically and the related morphism shall be noted $\varepsilon : \Pi_\Omega \to M(\Omega, \Omega)$ where M explicitly defined the representation. Particularly, this structure is algebraically speaking a join-semilattice $(S, \vee)$ with $S \subseteq \Omega$ where joining two elements, in that case, always return the same object, *i.e.* the representative object drawn from the block, then surely embodies the essence of the transitivity property.

3 Relational Encoding

Since the main purpose of the paper is to point out the (mis)matching of the relation model for partitions in the purpose of an SQL implementation, we need to elaborate encoding scenarios from partitions to relations.

We discuss only the encoding scheme introduced in the latter section and elaborated upon a tree-based representation model, *i.e.* membership encoding. In this section, we then review the *relational encoding schemes* for the set-theoretic model based on membership relations.

Despite this, we shall need for going further to map a membership encoded partition to the equivalence relations representation which enable us to emphasize operational problems that are shared with operations of Section 4.

3.1 Membership-encoding of a Partition

This encoding scheme represents the object-block membership relation within the relational model. It then requires a relation schema with 2 columns, one for the object, the other one for the block identifier. Block identifiers may be system generated, however we provide the block column with the same domain than the object column. It do not restrict the encoding capabilities since at worst, the singleton partition has size equal to the number of raw objects ($|P| = |\Omega|$), and it allows for effective query expressions (see Section 4).

Definition 1. *Given a partition P and its related equivalence relation $\sim_P$; assume a relation schema $M(\mathsf{elt} : \Omega, \mathsf{block} : \Omega)$ where columns* elt *and* block *are both objects of Ω. The relational encoding scheme ε of set partitions is defined as:*

$$\begin{array}{rccl} \varepsilon : & \Pi_\Omega & \mapsto & M(\Omega, \Omega) \\ & P & \to & \mathbf{I}(M) := \{(x, y) \mid x \in [y]\} \end{array}$$

In the above definition, we require equivalent classes $[y]$ of $\sim_P$ have all an *anchor* y, *i.e.* a highlighted object that identifies the all block. In conjunction with algebraic definition stated in Section 2.2, we arbitrarily decide to set the *minimum* object's value y as the anchor, assuming that the underlying set of $[y]$ has a deterministic and unique upper bound (integers are obviously totally ordered and match this condition). It is especially required for having a unique mapping $\varepsilon(P)$ for any input P. It follows that objects apart the root are all siblings and hence are not specifically ordered. Obviously, $(\mathbb{N}, \min(.))$ is a semi-lattice and $\min : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ follow the same identities as usual (associativity, commutativity and idempotency).

It should be noted that ϕ always returns an unique outcome for any entry, while gathering the anchors set does not of course provide a summary from an unique partition but for several ones.

3.2 From Membership to Equivalence and vice-versa

In the following, we discuss the transformation operation from one relational encoding scheme for partition P to the other, both ways. It may help showing the operational equivalence of both the encoding schemes.

Two issues have to be considered since a treatement is needed when we deal with lattice operators in membership encoding. Indeed, we have to consider whether or not, one M-block overlaps one N-block. In particular, applying the meet ($\Cap$) operator leads to compute explicitly the set-theoretic intersection between the two involves blocks and provide new blocks' id as much as is necessary whereas the join ($\Cup$) operator shall just overwrite objects blocks' id of only one of the two. Then, in order to compute a meet operation, we must ensure that all pairings between objects of both M-block and N-block are supplied and the corresponding equivalence relation should be available on the fly.

We Assume that P is encoded by $M(\mathsf{elt}, \mathsf{block})$;

A basic self-join on block is required to provide with the equivalence relation $\varphi(P) = E(x, y)$ and is very similar with $\phi(.)$ morphism earlier defined:

$$E := \sigma_{2=4}(M \times M)$$

We use here and in the following the *unnamed* flavor of RA defined by the set of operators $\{\sigma, \pi, \times, \cup, -\}$. Columns are denoted by their position into the relation. Size of RA expressions are then kept lower than with the named flavor.

The way back, from E to M requires much more thought in order to be implemented. Actually, it is given by the following *Domain Relational Calculus* (DRC) query:

$$M := \{(x, y) : \ E(x, y) \wedge \forall z.(E(x, z) \to z \geq y))\}$$

The DRC query is *safe* as far as we consider the usual *active domain semantics* where range values of a universally quantified variable are those from the actual values of both the database and the query. In both the M and E encoding

schemes, the active domain, denoted by **adom**, is the set of (id's) objects: $\mathbf{adom} := \text{adom}(M) = \text{adom}(E) = \pi_1(M)$.

It is worth to notice that the principle of the query relies on the assignment of the *anchor* value y to each set of objects that form a single block.

An SQL statement is required to practically perform the translation from E to M in R-DBMS. Fortunately, the basic DRC query above may be straightforwardly converted to an equivalent SQL statement by means of NOT EXISTS clause and subqueries. However, the formula expands to much more complex algebraic expression when translated to RA. Thus, query evaluation is costly and optimizations are not that easy since the query involves many joins.

We first rewrite the formula of the DRC query to provide an equivalent formula with only $\exists$, $\wedge$, $\vee$ and $\neg$ logical symbols:

$$M := \{(x, y) : \ E(x, y) \wedge \neg\exists z.(E(x, z) \wedge z < y)\}$$

The above formula is the relative complement in E of the following query:

$$\begin{array}{rcl} \bar{M} & := & \{(x, y) : \ E(x, y) \wedge \exists z.(E(x, z) \wedge z < y)\} \\ & \equiv & \{(x, y) : \ \exists z.(E(x, y) \wedge E(x, z) \wedge z < y)\} \end{array}$$

Then, the $\bar{M}$ relation could be expressed as:

$$\bar{M} := \pi_{1,2}(\sigma_{1=3,2=6,4=5}(E \times E \times \sigma_{1<2}(\pi_1(E) \times \pi_1(E))))$$

Finally, we are able to provide the RA expression of the translation "E to M" as follows:

$$\begin{array}{c} M := E - \pi_{1,2}(\sigma_{1=3,2=6,4=5}(E \times E \times \\ \sigma_{1<2}(\pi_1(E) \times \pi_1(E)))) \end{array}$$

Hence, the naive query evaluation requires 1 theta-join (with condition $1 < 2$), 3 equi-joins and a set difference operation over very large sets of tuples ($O(n^2)$, n being the number of objects) to be achieved.

4 Performing Operations

We focus in this section on the way to translate partition operators among $\{-, \cap, \Cap, \Cup\}$ within RA expressions over the membership-based encoding scheme ε which meet our requirements in terms of storage and computations.

We assume two partitions P and Q encoded *resp.* by relations M and N. For convenience, we do not distinguish relation schemes M and N from their respective instances. We also consider that P and Q are defined over the same ground set of objects Ω.

Example 1. *For instance, Table 1 represents a relational encoding of two partitions and shall be used to illustrate, step by step, the calculation of the operations.*

M	
elt	block
1	1
2	1
3	1
4	4
5	4
6	6

N	
elt	block
1	1
2	1
3	1
4	4
5	5
6	5

Table 1: Relational view of P and Q

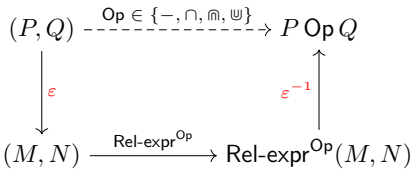


Figure 1: Relational Encoding Diagram

Figure 1 gives an overview of the challenge we are addressing in this Section. Main idea is to provide, for each partition operator Op, with a query expression $\mathsf{Rel\text{-}expr}^{\mathsf{Op}}$ in any relational language, such that $P\,\mathsf{Op}\,Q$ is given by:

$$P\,\mathsf{Op}\,Q = \varepsilon^{-1}(\mathsf{Rel\text{-}expr}^{\mathsf{Op}}(\varepsilon(P), \varepsilon(Q)))$$

4.1 RA expressible queries

Difference

The *Difference* operator in DRC is as follows:

$$\varepsilon(P - Q) := \{(x, y) :\ M(x, y) \wedge \forall z.\exists t.\neg\,(M(t, y) \leftrightarrow N(t, z))\}$$

where we build elt-block pairs (x, y) such that there is one such pair in M, and we can not find any block z in N that is equal to block y in M.

$\sigma_{2=4}(M \times N)$						$\varepsilon(P - Q)$	
1	1	1		...		4	4
2	1	1	⇒	4	4	4	⇒ 5	4
3	1	1		5	4	5	6	6
4	4	4		6	6	5		
5	4	5						
6	6	5						

Table 2: $\varepsilon(P - Q)$: First M-block matches first N-block

Example 2. *Given partitions P and Q from Table 1, principle of the difference operation is demonstrated on Table 2.*

As in Section 3.2, for RA translation purpose, the DRC formula should be first rewritten with $\exists$, $\wedge$, $\vee$ and $\neg$ logical symbols only:

$$\begin{aligned}\varepsilon(P-Q) := \{(x,y) : M(x,y) \wedge \\ \neg\exists z.\neg\exists t.((\neg M(t,y) \wedge N(t,z)) \vee \\ (M(t,y) \wedge \neg N(t,z)))\}\end{aligned}$$

We also observe that the above formula is the relative complement in M of the following query:

$$\begin{aligned}\{(x,y) : \exists z.(M(x,y) \wedge \neg\exists t.((\neg M(t,y) \wedge N(t,z)) \vee \\ (M(t,y) \wedge \neg N(t,z))))\}\end{aligned}$$

Then, we translate to RA each subexpression of the formula.

$$\begin{aligned}\neg M(t,y) \wedge N(t,z) &:= \sigma_{1=3}\left(((\mathbf{adom} \times \mathbf{adom}) - M) \times N\right) \\ M(t,y) \wedge \neg N(t,z) &:= \sigma_{1=3}\left(M \times ((\mathbf{adom} \times \mathbf{adom}) - N)\right) \\ R(x,y) \vee S(x,z) &:= R \cup S \\ \neg\exists x.R(x,y,x,z) &:= (\mathrm{adom}(R) \times \mathrm{adom}(R)) - \pi_{2,4}(R)\end{aligned}$$

where R and S are two generic relation names that match parts of the $\varepsilon(P-Q)$ DRC query. Remind that $\mathbf{adom} = \mathrm{adom}(M) = \mathrm{adom}(N) = \pi_1(M) = \pi_1(N)$ since support sets of both partitions P and Q are equal.

Thus, we are able to compose the entire algebraic expression from all those subparts:

$$\begin{aligned}\varepsilon(P-Q) := & M - \pi_{1,2}(\sigma_{2=3}(M \times ((\mathbf{adom} \times \mathbf{adom}) - \\ & \pi_{2,4}(\sigma_{1=3}(((\mathbf{adom} \times \mathbf{adom}) - N) \times M) \cup \\ & \sigma_{1=3}(((\mathbf{adom} \times \mathbf{adom}) - M) \times N)))))\end{aligned}$$

Here, we notice that the evaluation of the basic $\varepsilon(P-Q)$ query would require 3 true cross products, 3 equi-joins, 3 set differences and 1 union operation over relations of size in $O(n^2)$.

To follow on, since the equivalence $A \cap B \equiv A - (A - B)$ holds, then the set difference operator gives a proper definition for the *intersection* operator as well. We are also able to provide with a standalone definition for $\cap$ as the following DRC query:

$$\varepsilon(P \cap Q) := \{(x,y) : \ M(x,y) \wedge \exists z.\forall t.\,(M(t,y) \leftrightarrow N(t,z))\}$$

Observe that the above DRC query is semantically equivalent to the one where M and N have been permuted. The only difference is that block id's of the intersection would come from either M or N. It then makes possible further optimization within the query plan.

Meet

The *meet* operation $P \Cap Q$ is translated in DRC as:

$$\begin{aligned}\varepsilon(P \Cap Q) := \{(x,y) : \exists z.(M(x,z) \wedge M(y,z) \wedge \\ \exists t.(N(x,t) \wedge N(y,t) \wedge \\ \forall u.((M(u,z) \wedge N(u,t)) \rightarrow u \geq y)))\}\end{aligned}$$

Notice that the above formula conforms to the query pattern of the transformation from E to M as stated in Section 3.2. Indeed, assigning a single y value as an anchor to each distinct (M-block= z, N-block= t) pair is operationally similar to assigning the anchor values to blocks of equivalent objects. Actually, pairs (z,t) uniquely identify each block of the meet operator. It then requires to recompute the all equivalence relation from the membership-based encoding scheme.

Example 3. *From the partitions of Table 1, we can see on Table 3 the (M-block, N-block) pairs that identify each result block of the operation $\varepsilon(P \Cap Q)$. The next step to perform the meet operator is to assign anchors to blocks. It is shown on Table 4, from the self-join of* JTab $= \sigma_{2=4}(M \times N)$.

$\sigma_{2=4}(M \times N)$			
1	1		1
2	1		1
3	1		1
4	4		4
5	4		5
6	6		5

Table 3: $\varepsilon(P \Cap Q)$: (M-block, N-block) pairs

$\sigma_{(2,3)=(5,6)}$(JTab × JTab)					
1	1	1	1	1	1
2	1	1	1	1	1
2	1	1	2	1	1
3	1	1	1	1	1
3	1	1	2	1	1
3	1	1	3	1	1
4	4	4	4	4	4
5	4	5	5	4	5
6	6	5	6	6	5

$\Rightarrow$

$\varepsilon(P \Cap Q)$	
1	1
2	1
3	1
4	4
5	5
6	6

Table 4: $\varepsilon(P \Cap Q)$: objects & anchors

4.2 Datalog expressible query

The *Join* operation $P \Cup Q$ is not expressible in RA since a *transitive closure* needs to be performed [6]. Indeed, the union propagates to blocks each time there are

pairwise overlapping blocks from P and Q, until we reach a *fixpoint*. Since it is not possible to *a priori* plan the number of iterations in the propagation, then there is not any RA expression that could compute $\varepsilon(P \Cup Q)$.

However, Datalog is known to be a superset of RA that includes the required recursive part in the language. Thus, we may draw the Datalog program for the $\varepsilon(P \Cup Q)$ operation.

Basically, we decompose the query into 2 steps:

1. first, we build the connexions between block id's within one partition, and
2. second, we filter the result such that we stay with one single anchor for each set of equivalent block id's.

From this perspective, the join operation could be seen as the elaboration of an equivalence relation over the set of blocks themselves, followed by the anchor mechanism. The first step is not expressible within RA since it involves *reachability* issue within a graph, whereas the second step admits a RA expression.

Consider the graph $\mathcal{G}(V, \mathsf{E})$ where $V = \{x : \exists y.M(y, x)\}$ is the finite set of M-blocks and $\mathsf{E} = \{(x, y) : \exists z.\exists t.\exists u.M(z, x) \wedge M(t, y) \wedge N(z, u) \wedge N(t, u)\}$ makes a connexion (x, y) between M-block x and M-block y as far as there exist two objects z and t resp. in M-blocks x and y, that share the same N-block u. In other words, there is overlapping between M-block x and N-block u as well as between M-block y and N-block u.

It is worth to notice that we arbitrarily decided to build the graph over M-blocks, but M and N can permute.

As expected, $\mathcal{G}$ represents a binary relation, abusively denoted by $\mathcal{G}$, over M-blocks that is:

- *reflexive*: each M-block shares objects with itself;
- *symmetric*: if $(x, y) \in \mathsf{E}$, then $(y, x) \in \mathsf{E}$ since path from x to y by N-block u can be both ways;

The main goal of the first step is then to compute the transitive closure of relation $\mathcal{G}$. A pleasant side effect is that symmetry would be preserved and we obtain an *equivalence relation* θ over M-blocks at the end of the first step.

The Datalog program P that computes the transitive closure is as follows:

$$\begin{array}{lrcl} r_1: & \theta(y, y) & := & M(x, y) \\ r_2: & \theta(x, y) & := & \theta(x, z), M(t, z), N(t, u), N(v, u), M(v, y) \end{array}$$

Each Datalog rule is *safe* and *domain-independent*, and program P is trivially *stratified* since there is no negation. Recursivity is given by rule r_2 with θ both in the head and in the body of the rule. Ultimately, θ is an equivalence relation over M-blocks.

The second step of the process allows (a) to filter the tuples from θ and, (b) to provide with anchors to each equivalence class in θ and (c) to assign anchors

to objects. It can be defined by a DRC query as follows:

$$\varepsilon(P \Cup Q) := \{(x,y) : \exists z.(M(x,z) \wedge \theta(z,y) \wedge \forall t.(\theta(z,t) \rightarrow t \geq y))\}$$

We may have used Datalog to express that query as well, but the program would have been much more tricky than the concise DRC query.

To sum up, we provide with the sketch of the algorithm that performs the *Join* operation $\varepsilon(P \Cup Q)$, where we mix for convenience procedural loop and DRC queries in Algorithm 1. In the process, θ^{ω} usually denotes the fixpoint which is reached in a finite number of steps since $(\theta^{(i)})_i$ series is inflationist ($\theta^{(i)} \subseteq \theta^{(i+1)}$) and there is an upper bound on the size of θ^{ω} that is $|\mathbf{adom} \times \mathbf{adom}|$.

Algorithm 1 *Join* operation $\varepsilon(P \Cup Q)$

$\theta^{(0)} \leftarrow \{(y,y) : \exists x.M(x,y)\}$ ▷ Init step
repeat
$\theta^{(i+1)} \leftarrow \theta^{(i)} \cup \{(x,y) : \exists z.(\theta^{(i)}(x,z) \wedge \exists t.(M(t,z) \wedge \exists u.(N(t,u) \wedge \exists v.(N(v,u) \wedge M(v,y)))))\}$
$i++$
until $\theta^{(i+1)} = \theta^{(i)}$
$\theta^{\omega} \leftarrow \theta^{(i)}$
return $\{(x,y) : \exists z.M(x,z) \wedge \theta^{\omega}(z,y) \wedge \forall t.(\theta^{\omega}(z,t) \rightarrow t \geq y)\}$

The above analysis serves the purpose of an implementation of $\varepsilon(P \Cup Q)$ within regular R-DBMS. Knowing ANSI/ISO SQL3 introduces WITH RECURSIVE clause that extends RA features of SQL to mimic Datalog recursion, we are able to express a single SQL query as a *Common Table Expression* (CTE) statement to perform the $\Cup$ operation over partitions.

4.3 Optimizations through SQL specific features

To enhance further the relational encoding of partitions, we discuss in this section several options that apply to all or some of the operators. One straightforward idea to improve performance in query evaluation would be to use SQL extra capabilities that extend RA and therefore compute statistics so that it boost access methods and increase computation time for the relevant operators. However, since set partition operations are expected to be combined within complex expressions, then all the possible optimizations must be settled at query time and no side information (outside the encoding relations themselves) should be considered.

We then propose to explore two ways towards possible improvement in query execution.

Auxiliary Knowledge

The first optimization technique deals with the set-theoretic operators $-$ and $\cap$ applied within the membership-based encoding scheme ε. It basically consists in a pre-filtering step where blocks that cannot be part of the result set are early discarded.

Indeed, in the basic version of the SQL statement for set-theoretic operations $P \,\mathsf{Op}\, Q$, $\mathsf{Op} \in \{-, \cap\}$, each pair of ($M$-block, N-block) is checked in extension by scanning its all pairs of objects, to decide whether blocks are equal or not. Although this deep scan remains necessary for a few candidate pairs, it is possible to remove pairs (M-block, N-block) that do not satisfy coarse-grained conditions such like:

- minimum object values are equal;
- maximum object values are equal;
- size of blocks are equal.

Hence, it would be helpfull to define a *signature* of blocks, such like (min, max, count), that would act like a hashcode of the subset of objects in the block. The filtering step then requires to compare signatures pairwise.

Example 4. *Given* $P = 123|457|6|89$ *and* $Q = 123|467|58|9$*; we would like to perform* $P \cap Q$*. There are 12 pairs of (*M*-blocks,*N*-blocks) to compare. If we compute count values, then it remains 6 pairs only. With the minimum value, we stay with 2 candidate pairs:* $(123, 123)$ *and* $(457, 467)$*. The maximum value does not bring any further optimization here. Then, the refinement step would discard the second candidate pair to provide with the result set: 123.*

Then we provide with a revised version of the SQL statement that takes benefits from online fast computation of aggregates (min, max, count) to filter the pairs of (M-block, N-block). This alternative query cannot be expressed in RA.

Window Functions

The ANSI/ISO SQL3 introduces a nice feature called *Window Functions*, that is now part of almost all the R-DBMS, at least the "big four[2]". Roughly, the idea is to extend GROUP BY capabilities such that it is possible to assign each tuple of a table an aggregate value that depends on a customizable subset of tuples.

This feature can be useful for the *meet* operation $P \Cap Q$ within the ε encoding scheme. In that case, we would like to compute the minimum object value of each block and assign it to each tuple of that block. Thus, we build a window function within an SQL statement by partitioning the join $M \bowtie_{M.1=N.1} N$ on pairs (M-block, N-block). Since those pairs identify blocks of the result set,

[2]Oracle, Microsoft SQL Server, MySQL, PostgreSQL.

we could straightforwardly assign the minimum object value of the subsets of tuples w.r.t. that partitioning to each tuple of the join.

Obviously, window functions are not RA-expressible.

5 Experiments

In this section, we report and discuss experimental results. The main conclusion confirms what was expected from the considerations exposed in previous section, *i.e.* that the relational encoding essentially leads to query processing times that are unbearably high for other than small-size datasets. Since the very first objective is to try out relational encoding of partitions and their operations, the proposed strategies in Section 4.3 are suitable for corroborating our claim about partition data model mismatching within SQL framework. Each of them improve the overall performance for set partitions operators thanks to advanced features of SQL. Anyway, there is no way to overcome the closure computation into the SQL framework.

We shall supply further performance outcomes from a lean C++ implementation coined for the occasion which clearly outperforms SQL ones while complying with principles describes in Section 4. Obviously, such an implementation likely fails to closely imitate the data persistence layer and pagination process where data tables are too large to fit well in main memory. In such a case, we can implement a distributed ram management system which takes cares of splitting tables, if necessary, into independant storage unit and their retrieval through a distributed hash table.

As a consequence, we focus in this section on primary encoding of each operator proposed in Section 4, both into SQL queries and our C++ implementation, then operation-based optimizations introduced in Section 4.3, *i.e.* pre-filtering on blocks' id signature for $(-)$ operator and window function for $(\Cap)$ operator on both implementations.

Finally, we develop the same reasoning applied to relax the transitivity constraint to only consider a reachability constraint between blocks to be merged. We thus implemented a join $(\Cup)$ which relies on so-called Tarjan's Union-Find structure that keeps a block id's connected component instead of the whole clique as a spanning tree. Such a solution is likely to be implemented through an pointers' array-based layout since many both tree traversals and relinking are involved, and hence requiring several lookup in the associated table. This algorithm and those related, are usually implemented through this way to ensure their overall efficiency and as such, it seems clear that such methods cannot be push down easily into SQL. Concerning optimized version of $(\Cap)$ operator, we made an implementation similar to window functions mentioned above.

5.1 Settings

Experiments are conducted on randomly drawn partitions. Given two partitions P and Q, we assess the performance of $P \Cap Q$, $P - Q$ and $P \Cup Q$ only. Indeed,

the intersection $\cap$ relies on the set difference and is formulated as a relative complement to the difference. Hence, $\Cap$, $\Cup$ and $-$ are the legal baseline.

With the respect to the design of the SQL queries for each operator, empirical considerations were taken into account. In that purpose, we undertook several attempts to tune up the query optimizer according to available join methods (Nested Loop and its two scan method, Hash Join and Merge Join) in our R-DBMS. It turns out that if an improvement can be observed in few cases but also it may lead to a significant worsening for the same operator. Then, it seems that the default query plan computed by the optimizer is at last a good trade-off to achieve balanced performances.

We conducted experiments on a Windows XP (SP3) box powered by an Intel Q6600@2.4GHz CPU. We sent SQL queries into a PostgreSQL V9.1 R-DBMS. SQL statements are detailed in the Appendix A and some query plans are also provided as a mere example in the Appendix B.

5.2 Generating partitions

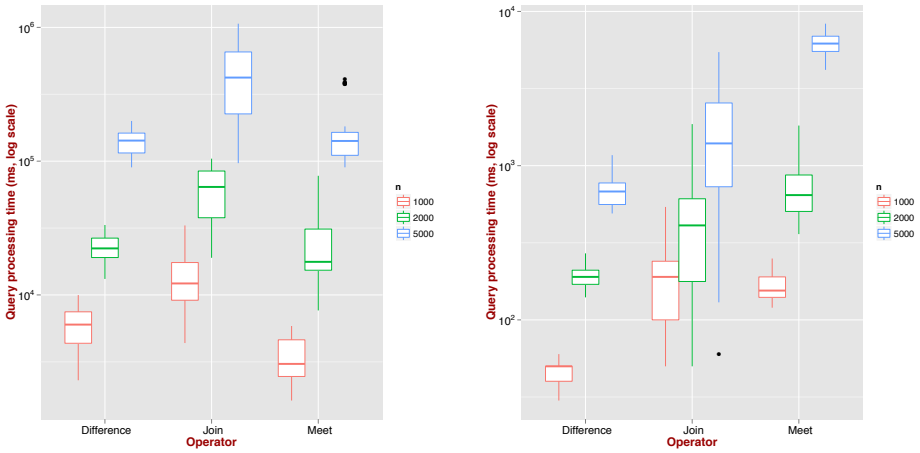
In the following, let $\mathsf{sort}(P) = \tau$ be the distribution of size of blocks within a partition P and τ is a decreasing sorted list where $\tau[i]$ gives the size of block a_i in P. Given n objects, we first draw partition P, then generate Q by applying random permutations on P. The sort of raw partition P follows a power law. Indeed, a remarkable property of many natural or man-made phenomena is that, given a population, the frequency of its subpopulations may very often be well modeled by a *power law* [4]. Generation of such partitions is easily achieved with the Chinese Restaurant (stochastic) Process (CRP) [8]. The CRP draws a random partition over the set of integers $[1...n]$. As a noteworthy property, the underlying distribution, known as the *Ewens distribution*, is said to be exchangeable, *i.e.* the probability of a partition only depends on block sizes. The expected number of blocks k grows as $O(\alpha \log n)$, where α is the scale parameter of the CRP.

Next, some random permutations are performed on P to generate a list of partitions $(P^{(1)}, \ldots, P^{(\ell)})$ such that:

- $d(P, P^{(i)}) < d(P, P^{(i+1)})$ is consistent w.r.t. a distance-based function $d : \Pi_\Omega \to \mathbb{N}$ that evaluates the required number of permutations on objects from P to reach its "noisy" counterpart $P^{(i)}$;
- Each $P^{(i)}$ still follows the raw P distribution ($\mathsf{sort}(P^{(i)}) \equiv \mathsf{sort}(P^{(0)})$) thanks to permutations $\sigma : a_j \mapsto a_k$ that preserve $\tau^{(i)}[j] = \tau^{(i+1)}[k]$;
- At last, $\forall i$, $\mathsf{sort}(P^{(i)}) \equiv \mathsf{sort}(P^{(i+1)})$.

The whole generation process can be summarized as follows:

1. Init: random choice of a linear ordering on a subset of blocks $\{a_0, \ldots, a_\ell\} \subseteq P$;
2. Loop $0 \leq i \leq \ell - 1$: permutation of $\min(\tau[i], \tau[i+1])$ objects in the pair of adjacent blocks (a_i, a_{i+1}) to build $P^{(i+1)}$.



(a) SQL (b) C++

Figure 2: Processing time for $n = 1\,000, 2\,000, 5\,000$ (from left to right for main operators $-$,⩒,⩓).

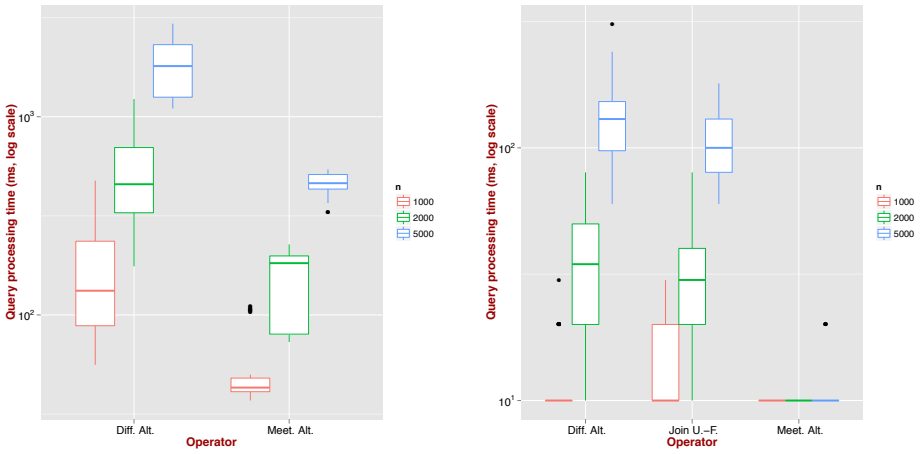
Such modeling allows to accurately monitor partition operations according to some sequential changes applied on operands, and at last, infer some properties that may impact performance. We shall note also that the equalities $\mathrm{rk}(P^{(0)}) = \mathrm{rk}(P^{(1)}) = \ldots = \mathrm{rk}(P^{(\ell)})$ always hold

5.3 Results and analysis

Results are reported for partitions P and Q defined over $n = 1\,000$, $2,000$ and $5\,000$ objects. Although those numbers are quite low, we observed in experiments that query processing times preclude increasing n by a further order of magnitude. The number of blocks is set to range between 10 and 20. Overall, the largest blocks, generated under the CRP mechanism, are typically about half the size of the support set, while for $n = 2\,000$ and $n = 5\,000$, there are a few blocks that are singletons.

We report query processing time related to SQL implementations in Fig. 2a,3a,4 while C++ ones are depicted in Fig. 2b,3b. The boxplots describe the observed variability from a set of experiments, where:

- The number of blocks ranges in [10..20];
- The number of random permutations applied on P to generate Q varies from 140 to 466 for $n = 1\,000$, from 184 to 532 for $n = 2\,000$, and from 406 to 1 728 for $n = 5\,000$.



(a) SQL (b) C++

Figure 3: Processing time for $n = 1\,000, 2\,000, 5\,000$ (from left to right for optimized version of operators $-$, $\Cup$ (C++ only) and $\Cap$).

Concerning genuine operators, the $P \Cap Q$ operation is more expensive, though slightly, than $P - Q$ whereas $P \Cup Q$ computation performs worst than all others. Besides, operations on partitions of 5 000 objects are all much more costly (one order of magnitude) than those with 1 000 and 2 000 objects.

Roughly, adding noise into partitions yields to increase mean squared of the execution time. Ultimately, P op $P^{(\ell)}$, $\mathsf{op} \in \{\Cap\ \Cup\ -\}$ shows outlier runs w.r.t. the execution time, where $P^{(\ell)}$ is the farthest partition from P in the generation process. This observation is still emphasized with the growing size n of the support set. The very first conclusion is that query processing time rapidly becomes prohibitive, even for data sets with moderate size.

Moreover, optimized versions of meet and difference operator perform much better than these others, it is nevertheless necessary to take into account that this do not outweigh the poor performance of both genuine operators. Indeed, the comparison with our C++ implementation in terms of execution time is a conclusive outcome since the imperative version is ultimately faster than the SQL version, even without use of any particular memory manager/allocator (since ground set is sufficiently small to fit in main memory).

It shall be then analyzed if optimized operations could be seen as a sufficient basis to ensure the viability of complex expression computation. Following this outline, (Fig. 4) depicted several independant trials where we measured execution time of an increasing sequence of optimized ($\Cap$), initially using only 2 operands until reach 10 ones, randomly choosen among the collection $(P^{(0)}, P^{(1)}, \ldots, P^{(\ell)})$.

Furthermore, it is clear that there is by no means any performance worsening, but also improvement, while a new operand is appended to the calculus, throughout every trial we ran. It makes sense therefore that the behavior of each trial was not affected in either ways by the query optimizer as opposed to algebraic properties of our collection of partitions.

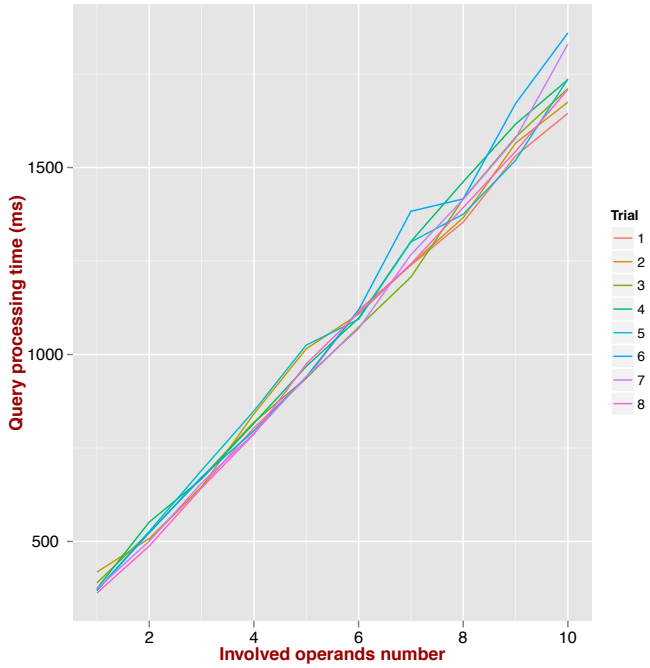


Figure 4: SQL query processing time for increasing sequence of combination of (windowed) $\Cap$ operations with $n = 5\,000$

Indeed, because their rank are mutually equal, it implies that the result of any meet sequence should lead to a partition strictly ranked below in the lattice. As we constrained each instance to be increasingly far to the very first generated partition, it comes naturally than the expected result of any trial at any time should have been the computation between the two farthest involved partition, *i.e.* if we sample in $\{P^{(0)}, \ldots, P^{(\ell)}\}$ in order to get an ordered sequence with earliest $P^{(i)}$ and lastest $P^{(i+k)}$, any computation shall be equivalent to make $P^{(i)} \Cap P^{(i+k)}$. Computing efficiently a sequence of the same lattice operator leads hence to find the cheapest query among subsets of instances which promptly reach the first common projector in the lattice (towards either the $\top$ or the $\bot$), and that clearly underlies the need of a well-designed query plan and therefore it is at least a challenging task.

6 Discussion

Althought relational modelling remains the most commonly used method, some alternative ways grow day to day due to its lack to cope with large amounts of data. For instance, we could have used the nested relational algebra while we do not since our membership encoding precludes the need to compute both nest and unnest operators. Considering also that every membership encoded partition can be translated as an hypergraph representation whose each hyperedge contains $\tau[i]$ vertices for each block a_i of the genuine partition. Such a representation could be handle through a graph-based management system and *ad hoc* algorithms. However, the purpose of such a management system is to deal with general hypergraph and related query optimizer should not be regarded as a dedicated solution to compute algebraic operations on partitions and therefore inside complex expressions combining these ones.

7 Conclusions

In this paper, we provide a contribution towards achieving some relational modeling of partition through several encoding scenarii, so that they can handle set partitions of a collection of objects. This is typically needed by large-scale repositories storing both data and results of data analysis, or data mining tasks which take partitions as inputs. For that purpose, we disclosed two relational encoding methods of a set partition through an object-block membership relation and the more usual equivalence relation. We also studied their respective computing framework. We then translated each operator of both partition lattice and algebra of sets as relational algebra queries, wherever possible, Datalog query otherwise. We gave a few sketches to enhance the behavior for some operators through storage of additional informations on-the-fly. Through several experimentations, we showed that computing operators over partitions is globally intractable when their underlying ground set is growing. Even if we consider SQL-based optimization that does not allow for the avoiding to design a complete management system to handle set partitions, not only restricted to those defined with the same ground set, which should be designed in order to fully unleash the potential of their inherited canonical, that is algebraic, properties by means of an *ad hoc* management system.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the xml-to-relational mapping problem. In *Proc. of the 6th ACM Int. Workshop on Web Information and Data Management (WIDM'04)*, 2004.

[3] M. Carey, J. Kienan, J. Shanugasundaram, E. Shekita, and S. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *Proc. of the 26th Int. Conf. on Very Large Databases (VLDB'2000)*, 2000.

[4] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, Nov. 2009.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[6] G. Dong, L. Libkin, and L. Wong. Local properties of query languages. *Theor. Comput. Sci.*, 239:277–308, May 2000.

[7] L. M. G. Feijs and R. C. van Ommering. Relation partition algebra — mathematical aspects of uses and part-of relations. *Science of Computer Programming*, 33(2):163–212, 2 1999.

[8] S. Goldwater, T. L. Griffiths, and M. Johnson. Producing power-law distributions and damping word frequencies with two-stage language models. *J. Mach. Learn. Res.*, 999999:2335–2382, July 2011.

[9] T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 109–120, New York, NY, USA, 2002. ACM.

[10] T. Halpin and T. Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2008.

[11] T. Halverson and A. Ram. Partition algebras. *Eur. J. Comb.*, 26:869–921, August 2005.

[12] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[13] W. Keller. Mapping objects to tables: A pattern language. In *Proceedings of EurPLoP*, 1997.

[14] H.-P. Kriegel and A. Zimek. Subspace clustering, ensemble clustering, alternative clustering, multiview clustering: What can we learn from each other? In *Proc. 1st Int. Wksp MultiClust 2010 w/ 16th ACM SIGKDD Conf. KDD 2010*, Washington, DC, USA, 2010.

[15] K. K. Pu and A. O. Mendelzon. Concise descriptions of subsets of structured sets. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'2003)*, pages 123–133, 2003.

Appendix A

In this appendix, we provide with the SQL statements for each operator in $\{-, \cap, \Cap, \Cup\}$. Every query assume there exist 2 membership-based encoding tables M and N for partitions P and Q defined over the same support set. Remind that relation schemes are $M(\mathsf{elt}, \mathsf{block})$ and $N(\mathsf{elt}, \mathsf{block})$. object id's (a) are unique within one table, (b) are shared among both tables M and N, and (c) must be totally ordered. Without loss of generality, we declared elt columns of *positive integer* type to fulfill the requirements.

```
SELECT outM.elt,
       outN.block
FROM $1 outM JOIN $2 outN USING (elt)
WHERE EXISTS (
    ( SELECT *
      FROM $1 inM JOIN $2 inM USING (elt)
      WHERE outM.block=inM.block AND
            outN.block<>inN.block)
  UNION
    ( SELECT *
      FROM $1 inM2 JOIN $2 inN2 USING (elt)
    WHERE outM.block<>inM2.block AND
          outM.block=inN2.block));
```

Listing 1: Difference operation

```
SELECT outM2.elt,
       foo.min AS block
FROM
( SELECT block, min(elt), count(elt)
  FROM $1 outM GROUP BY block
) AS foo
JOIN
( SELECT block, min(elt), count(elt)
  FROM $2 outN GROUP BY block
) AS bar
USING (min, count)
JOIN $1 outM2 ON (outM2.block=foo.block)
WHERE EXISTS (
    ( SELECT *
```

```
      FROM $1 inM JOIN $2 inN USING (elt)
      WHERE foo.block=inM.block AND
            bar.block<>inN.block)
    UNION
    ( SELECT *
      FROM $1 inM2 JOIN $2 inN2 USING (elt)
      WHERE foo.block<>inM2.block AND
            bar.block=inN2.block));
```

Listing 2: Difference operation (alt.)

Intersection operation was implemented the same way than the Difference (2 flavors), except the EXISTS clause turned into NOT EXISTS. Thus, we do not copy-paste the listing here.

```
SELECT foo.elt AS elt,
       bar.elt AS block
FROM
( SELECT elt,
  outM.block AS block1, outN.block AS block2
  FROM $1 outM JOIN $2 outN USING (elt)
) AS foo
JOIN
( SELECT elt,
  outM2.block AS block1, outN2.block AS block2
  FROM $1 outM2 JOIN $2 outN2 USING (elt)
) AS bar
ON (foo.block1=bar.block1 AND foo.block2=bar.block2)
WHERE NOT EXISTS (
  SELECT *
  FROM $1 inM JOIN $2 inN USING (elt)
  WHERE inM.block=foo.block1 AND
        inN.block=foo.block2 AND
        elt<bar.elt);
```

Listing 3: Meet operation

```
SELECT outM.elt,
       min(elt) OVER (
           PARTITION BY (outM.block,
                         outN.block))
FROM $1 outM JOIN $2 outN USING (elt);
```

Listing 4: Meet operation (alt.)

```
WITH RECURSIVE
BTC(bfrom, bto) AS (
    ( SELECT mb,
             mb
    FROM BJoin )
    UNION
    ( SELECT t.bfrom,
             j2.mb
    FROM BTC t
      JOIN BJoin j1 ON (t.bto=j1.mb)
      JOIN BJoin j2 ON (j1.nb=j2.nb))),
BJoin (mb, nb) AS (
    SELECT inM.block,
           inN.block
    FROM $1 inM JOIN $2 inN USING (elt))
SELECT outM.elt,
       t.bfrom AS block
FROM $1 outM JOIN BTC t ON (outM.block=t.bto)
WHERE NOT EXISTS (
```

```
20      SELECT *
21      FROM $1 inM2 JOIN BTC t2 ON (inM2.block=t2.bto)
22      WHERE inM2.elt=outN.elt AND
23              t2.bfrom < t.bfrom);
```

Listing 5: Join operation

Appendix B

We provide also some raw execution query plans for the following operators $\{-, \Cup\}$ that shall illustrate the complexity entailed in the various stages leading to their computation. Both terms (*i.e.* partitions in their membership-based encoding), that are parameters of the query, are defined on a ground set which contains exactly 5000 objects whereas one partition is obtained by noising the second one. The first assesment is the use of indexes during processing of merge conditions on blocks id's. It is right that even though we did not want to use them, using integer representation for both objects and blocks id's naturally implies the creation of an implicit index by the DBMS. It leads to incredibly speed up the processing of a query whereas their use cannot be extended to support queries where some bindings are made on partial results provided by some subqueries. The relatively low number of objects in the ground set leads further to compute directly both sort and hash operations in memory. Despite a favourable experimental environment to measure ease of computing operators, there is a lack of good results to assess such representations into SQL queries.

exec. time	rows	node
11119	3552	Nested Loop Anti Join — Join Filter: (((outM.block = inM.block) AND (outN.block <> inN.block)) OR ((outM.block <> inM2.block) AND (outM.block = inM2.block)))
10	5000	Merge Join — Cond: (outM.id = outN.id)
3.2	5000	Index Scan using $1_elt_key on $1 outM
6	5000	Index Scan using $2_elt_key on $2 outN
7535	3740	Materialize
10	5000	Hash Join — Cond: (inM.elt = inN.elt)
4.7	5000	Seq Scan on $1 inM
4.8	5000	Hash Buckets: 1024 Memory Usage: 137kB
4.9	5000	Seq Scan on $2 inN

Table 5: Query plan for difference operation

exec. time	rows	node
1488	5000	Hash Anti Join — Cond: (p1.elt=p2.elt) — Join: (t2.n<t.n)
		CTE cjoin
11.3	5000	Hash Join — Cond: (p1.elt=p2.elt)
4.5	5000	Seq Scan on $1 p1
6.2	5000	Hash Buckets: 1024 Mem.Usage: 137kB
5.5	5000	Seq Scan on $2 p2
		CTE t
2857	32	Recursive Union
39	5000	CTE Scan on cjoin
6754	1290969	Merge Join — Cond: (t.p = j1.n)
0.1	8	Sort — Key: t.p — Method: quicksort Mem: 17kB
0.02	8	WorkTable Scan on t
9141	2463782	Materialize
12281	2338475	Sort — Key: j1.n — Method: extern. merge Disk: 41mB
8498	2339340	Merge Join — Cond: (j1.p = j2.p)
22	5000	Sort — Key: j1.p — Method: quicksort Mem: 324kB
11.2	5000	CTE Scan on cjoin j1
4003	2339328	Sort — Key: j2.p — Method: quicksort Mem: 324kB
4.35	5000	CTE Scan on cjoin j2
21.2	9344	Merge Join — Cond: (p1.block = t.p)
12.4	5000	Sort — Key: p1.block — Method: quicksort Mem: 324kB
4.2	5000	Seq Scan on $1 p1
6.7	9332	Materialize
0.07	32	Sort — Key: t.p — Method: quicksort Mem: 18kB
43612	32	CTE Scan on t
129	9344	Hash Buckets: 4096 Batches: 262144 Mem. Usage: 1kB
12	9344	Merge Join — Cond: (p2.block = t2.p)
4.9	5000	Sort — Key: p2.block — Method: quicksort Mem: 324kB
2.2	5000	Seq Scan on $1 p2
3.9	9332	Materialize
0.04	32	Sort — Key: t2.p — Method: quicksort Mem: 18kB
0.02	32	CTE Scan on t t2

Table 6: Query plan for join operation