A First Attempt to Computing Generic Set Partitions: Delegation to an SQL Query Engine

Frédéric Dumonceaux, Guillaume Raschia, Marc Gelgon

A First Attempt to Computing Generic Set Partitions: Delegation to an SQL Query Engine

Frédéric Dumonceaux, Guillaume Raschia, and Marc Gelgon

LINA (UMR CNRS 6241), Université de Nantes, France
firstname.lastname@univ-nantes.fr

Abstract. Partitions are a very common and useful way of organizing data, in data engineering and data mining. However, partitions currently lack efficient and generic data management functionalities. This paper proposes advances in the understanding of this problem, as well as elements for solving it. We formulate the task as efficient processing, evaluating and optimizing queries over set partitions, in the setting of relational databases. However, producing universally fast execution plans remains a challenging task, since the underlying relational model has a significant impact on the algebraic definition of the operators and therefore on their implementation in terms of space and time costs.
We first demonstrate that there is no trivial relational modeling for managing collections of partitions. We formally motivate a relational encoding and show that one cannot express all the operators of the partition lattice and set-theoretic operations as queries of the relational algebra. We investigate SQL features beyond FO to build optimized queries for partition operators. We provide multiple evidence of the inefficiency of FO queries. Our experimental results enforce this evidence, event accounting for careful SQL query optimization.
We claim that there is a strong requirement for the design of a dedicated system to manage set partitions, or at least to supplement an existing data management system, to which both data persistence and query processing could be delegated.

1 Introduction

Let us consider a set of items organized into disjoint groups. If these groups cover together the set of items, they form a partition of this set. Partitions are a very common and useful data structure manipulated in data engineering and mining. However, this structure is not yet a *first-class citizen*, in terms of data management functionalities. This becomes particularly critical when scaling up to arbitrary partitions defined over a very large set of items. As a motivating vignette, one may think of a research activity conducting exploratory analysis via unsupervised classification on massive scientific data in a collaborative fashion. Researchers would then supply their clustering results to a shared repository and try to elaborate forms of comparison and combination of partitions, for instance, to acquire knowledge about the all experiments. Our perspective is that further

exploitation of these results would gain from the availability of (a) independence of the physical and logical representation of set partitions b.t.w. of an abstract layer (b) query facilities and (c) an optimized set partition query engine.

In this paper, we address the issue of querying set partitions regardless of types, features and any other auxiliary information of the items. In other words, given a set Ω of items, our very basic assumption states that, given two distinct items x and y in Ω, we know whether they are in the same group ($x \in [y]$) or not. One does not make any other assumption on values of x and y, such like having identical or similar features. This *agnostic* position, being the most generic approach, makes possible combining partitions of the same set Ω of items with very different points of view (aka. feature spaces) w/o any trade-off or tricks to *a priori* merge both representations.

As a motivating example, we shortly present hereunder a generic application scenario where storing and querying partitions can greatly support data exploration task.

Scenario: Set Partitions for an OLAP Cube-like Structure In declarative query languages to multidimensional data $\Omega : \mathcal{A}$, coined tables, a partitioning task is usually performed explicitly by picking attributes from $\mathcal{A}$ and designing an *aggregation mapping* $\texttt{agg} : 2^{\Omega} \times 2^{\mathcal{A}} \rightarrow 2^{2^{\Omega}} \times \mathbb{R}$, for which one wants to group tuples from Ω and ultimately get an aggregate value in $\mathbb{R}$ such that it underlies a partition of the ongoing table. As an example, we may think about the SQL `group-by` clause over a subset X of attributes in $\mathcal{A}$, with any usual aggregate function (min, max, sum, count, avg). In this setting, every pair of tuples $(t, u) \in \Omega^2$ such like $t[X] = u[X]$ belongs to the same group and the aggregate function computes one single value for each group. By the way, the aggregation mapping yields to a disjoint assignment of tuples into groups and then, it denotes a partition ($P_{\texttt{agg}} \in 2^{2^{\Omega}}$) of the whole table.

Aggregation operators are the core of *online analytical processing* (OLAP) [9]. Roughly, with a slight generalization over the aggregate value, that depends on a given numerical attribute called *measure*, we can basically model cubes with the previous aggregation mapping. In real-life settings, cubes have only 3 *dimensions* in $\mathcal{A}$ for visualization purpose as in (product, customer, area) with the sold-amount measure that can be sum up, for instance. Here, dimensions are multi-scaled such that we can switch from quarters (zip code) to cities and consider a new—smaller—set of attribute values to map to new—coarse-grained—groups. This operation is usually coined *roll-up* and its dual *drill-down* in the cube. They can easily be translated to set partition refinement in our model for aggregation mapping. Then, it seems to be a relevant solution to build statistical databases and aggregate summaries at large. One could gain also benefit from saving several frequent aggregation queries as set partitions along with their attribute vectors to ease further analysis as soon as one wishes to design new queries from the former ones [16].

Problem Scope and Contributions The above setting illustrates the need for management systems to support manipulations on set partitions. The baseline work direction consists in carrying out the mapping of partitions onto the well-established Codd's *relational model* [5]. This indeed provides mature systems with sophisticated query capabilities, query optimization policies and well-founded theoretical background. We then explore the relational encoding of set partitions as well as delegation of operators on partitions to the SQL engine.

Further Reading... The remainder of the paper is organized as follows. Section 2 presents the basic mathematical background required to operate on set partitions. Different flavors are provided. We also introduce the encoding schemes for partitions. In Section 3, we express relational queries against encoding schemes that relate to a collection of set partition operations. Section 4 reports experimental results that confirm our static analysis. Sections 6 and 7 respectively cover related work and draw concluding remarks.

2 Data Model(s)

This section motivates the relational encoding, formally defined in Section 2.3. The data model actually depends on combinatorial properties of set partitions. Set partitions admit two levels of nesting and constraints: a partition $P = \{a_1, a_2, \ldots, a_n\}$ is indeed a set of sets (blocks a_i) of items where blocks satisfy both $\bigcup_{a_i \in P} a_i = \Omega$ and $a_i \cap a_j = \emptyset$ if $a_i \neq a_j$, with respect to the ground set Ω of items. For ease of reading, we shall denote a partition as $P = \diamondsuit\spadesuit\heartsuit\clubsuit \mid \infty\neg \mid \natural\sharp\flat$ where $\mid$ separates blocks of items. Thereafter, we use natural numbers $\mathbb{N}$ as the underlying domain for Ω without loss of generality.

2.1 Partition Algebra

The set of all partitions Π_Ω defined on the same ground set Ω is endowed with the *refinement relation* $\leq$ so that the *poset* $(\Pi_\Omega, \leq)$ is the well-known *partition lattice*. In this lattice, P *refines* Q, denoted $P \leq Q$, if and only if every block of P is a subset of a block in Q.

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

$$P \vee Q := \inf \sup \{R \mid P \leq R \text{ and } Q \leq R\}$$
$$P \wedge Q := \sup \inf \{R \mid R \leq P \text{ and } R \leq Q\}$$

Example 1. Given $P = 123|456|78|9$ and $Q = 123|45|67|89$, then $P \wedge Q = 123|45|6|7|8|9$ et $P \vee Q = 123|456789$.

This leads to the following identity rule: $(P \leq Q) \iff (P \vee Q = Q) \wedge (P \wedge Q = P)$ and $\top_\Omega \leq P$ and $P \leq \bot_\Omega$ for all $P, Q \in \Pi_\Omega$, where $\bot_\Omega$ denotes the

bottom partition where each block contains a single object and $\top_\Omega$ is the *top* partition with a single block.

Besides, as partitions are also a family of sets, we would like to consider some set-theoretical operators which apply to blocks. Indeed, the very first idea behind partitions is that they are subsets of the *boolean lattice* 2^Ω. Further, we can apply some boolean operations on pairs of blocks:

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

The 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.

Example 2. Following Example 1, $P - (P \wedge Q) = 456|78$ since 123 and 9 are blocks of $P \wedge Q$. Support set of $P - (P \wedge Q)$ becomes $\mathcal{S} = \{45678\} \subseteq \Omega$.

Finally, the set union operation $\cup$ is not expected to preserve mutual disjunction since blocks of distinct partitions may overlap, then it is not eligible in our study. To sum up, we are staying with four set partition operations: meet ($\wedge$), join ($\vee$), intersection ($\cap$), difference ($-$), that we would like to perform on generic set partitions.

2.2 Extensional Representation

Let the classical set-theoretic representation of a partition be called the *intension*. In this conceptual view, blocks are anonymous subsets of items. For operational purpose, there is a requirement for elaborating an encoding schema that is able both to perform boolean tests over blocks (membership, containment) and to update block configurations. Then, as a counterpart to the intentional view, *extension* refers to an equivalent representation, where the same properties hold. It must basically preserves item-to-block membership. And ultimately, we are concerned about the design of a one-to-one mapping ε from intention to extension of partitions, which conveys the same structure within each representation. To this end, in this section we go from the underlying item-to-item associations up to the relational encoding schema.

Partition Decomposition An 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). This extension simply emphasizes that, given two items $x, y \in \Omega$, x and y are in the same block of partition P iff $x \sim_P y$. Let us define the morphism $\phi : P \mapsto \sim_P$ as:

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

We may retrieve the original partition through its inverse $\phi^{-1}(\sim_P) := P/\sim_\Omega$ and hence recover explicit item assignments to their respective blocks.

Such an extensional representation is the most expensive description as its space complexity is in $O(|\Omega|^2)$. Actually, it provides the all item-to-item associations of a partition. For instance, $\phi(1234|56) = \{(11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43, 44, 56\}$. To follow on the intensional-to-extensional mapping, it is worth to notice that the following properties hold:

$$\phi(P \wedge Q) = \sim_P \cap \sim_Q$$
$$\phi(P \vee Q) = \langle \sim_P \cup \sim_Q \rangle$$

One requires an algebraic closure operator $\langle \mathcal{R} \rangle = \bigcap \{\sim \subseteq \Omega \times \Omega \mid \mathcal{R} \subseteq \sim\}$ in order to "promote" $\sim_P \cup \sim_Q$ as a set partition.

At this stage, we come up with a—very inefficient—extension b.t.w. of the equivalence relation counterpart of a partition. In this setting, performing a set-theoretic operation over partitions requires round tripping from the scope of the extensions to intentions and backward. This mechanism is described in the following equality:

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

Then, a key issue for an efficient encoding is to avoid such intensive enumeration of item-to-item associations while retrieving explicit item memberships when performing set operators $\{\cap, -\}$.

Tree-based Representation Thankfully the *transitivity* property of the equivalence relation can be relaxed in our setting to a simple *reachability* property. From a graph-theoretical perspective, we may consider as consistent the fact that x, y, z are equivalent *w.r.t.* a partition if there is a chain that only guarantees existence of a strongly connected component rather than a complete (sub)graph.

Such a structure is optimal according to space complexity if it is a minimum spanning tree, *i.e.* $O(n)$-space, which entails every item with at least two edges (except for the leaves) and states that every item within a block is reachable by all the others. As an example, partition $1234|56$ can easily be represented by $\{12, 23, 14, 56\}$ as a forest of two trees rooted resp. by 1 and 5. Furthermore, this tree-based representation can straightforwardly be traversed through a relational encoding since it is utmostly flattened.

2.3 Relational Encoding

As previously stated, the relational model natively supports partitions and provides a very straightforward encoding scheme for the tree-based representation of generic set partitions. We refer to the relational mapping as a *membership* encoding scheme.

This encoding scheme represents the item-to-block membership within the relational model.

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

Fig. 1: Relational view of $P = 123|45|6$ and $Q = 123|4|56$

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 items of Ω. The relational encoding scheme ε of set partitions is defined as:*

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

In the above definition, we require each equivalent class $[y]$ of $\sim_P$ to have an *anchor* y, *i.e.* a highlighted item that identifies the block. In conjunction with algebraic properties stated in Section 2.2, we arbitrarily decide to set the *minimum* item's value y as the anchor, assuming that $[y]$ has a deterministic and unique lower bound while the Path Independence condition [13] entails that ε must be defined as an extremal function, and thereafter, y could be assigned min or max value only. It is especially required for having a unique mapping $\varepsilon(P)$ for any input P. It follows that items, except the anchor, are all siblings and the underlying tree-based representation is actually a *star*. Hence objects but the root of a block are not ordered.

Example 3. Tables M and N in Figure 1 show the relational mapping of partitions $P = 123|45|6$ and $Q = 123|4|56$. Block identifiers, also known as anchors, are 1,4,6 for P and 1,4,5 for Q. They all meet the minimum value of each block. Each block, such as 123, is encoded by a set of edges, e.g. $\{(11), (12), (13)\}$, such that it builds a star with the anchor as the root, and follows the tree-based representation.

3 Performing Operations

We focus in this section on the way to translate partition operators among $\{-, \cap, \wedge, \vee\}$ within relational queries over the membership encoding scheme ε.

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.

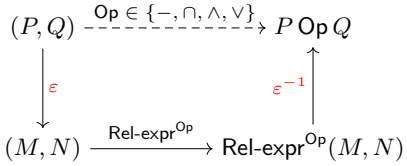

Fig. 2: Relational Encoding Diagram

Figure 2 gives an overview of the challenge we are addressing in this Section. The main idea is to provide, for each partition operator Op, a query expression $\mathsf{Rel\text{-}expr}^{\mathsf{Op}}$ in any relational language, such that $P\,\mathsf{Op}\,Q$ is given by:

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

More precisely, three main issues have to be addressed to deal with set partition operators in the membership encoding : (i) block *comparison*, (ii) block *intersection* and (iii) block *identification*.

Firstly, set-theoretic operators $\{-, \cap\}$ on partitions require testing equality of M-blocks and N-blocks pairwise. We also have to decide whether or not, a given M-block overlaps an N-block, especially to perform the join ($\vee$) operation that merges overlapping blocks. Those are block comparison operations, or predictates on blocks.

Secondly, performing the meet ($\wedge$) operator involves calculating the set-theoretic intersection of pairs of M-blocks and N-blocks. The baseline mechanism requires computing the equivalence relation $\phi(P)$ *resp.* $\phi(Q)$ from M *resp.* N at query time. It is easily formulated as $\pi_{1,3}\sigma_{2=4}(M \times M)$ but it introduces severe performance pitfall (see Section 4).

Last but not least, every lattice operation (meet and join) requires the assignement of an anchor (block id) to the newly created blocks. Therefore, the process must iteratively fix block id's to minimum values.

3.1 Difference

The *Difference* operator in *Domain Relational Calculus* 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.

We then come up with the following algebraic expression:

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

where **adom** = adom(M) = adom(N) = $\pi_1(M) = \pi_1(N)$ since support sets of both partitions P and Q are equal. Hence, the naive evaluation of $\varepsilon(P - Q)$ query would require 3 full cross products, 3 equi-joins, 3 set differences and 1 union operation over relations of size in $O(|\mathbf{adom}|^2)$.

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.

3.2 Meet

The *meet* operation $P \wedge Q$ translates in DRC as:

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

The above formula combines block intersection issue with block identification issue. Indeed, assigning a single y value as an anchor to each distinct (M-block= z, N-block= t) pair is operationally similar to assigning an anchor value to a set of equivalent objects. Actually, pairs (z, t) uniquely identify each block of the meet operator. It then requires recomputing the all equivalence relation from the membership encoding scheme.

Example 4. From the partitions of Figure 1, we can see on Table 1 the (M-block, N-block) pairs that identify each result block of the operation $\varepsilon(P \wedge Q)$. The next step to achieve the meet partition is to assign anchors to blocks. It is shown on Table 2, 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 1: $\varepsilon(P \wedge Q)$: (M-block, N-block) pairs

$\sigma_{(2,3)=(5,6)}(\text{JTab} \times \text{JTab})$

1	1	1	1	1	1
1	1	1	2	1	1
1	1	1	3	1	1
2	1	1	1	1	1
2	1	1	2	1	1
2	1	1	3	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

$\Rightarrow$

$\varepsilon(P \wedge Q)$

1	1
2	1
3	1
4	4
5	5
6	6

Table 2: $\varepsilon(P \wedge Q)$: objects & anchors

3.3 Join

The *Join* operation $P \vee Q$ 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 $\varepsilon(P \vee Q)$.

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 (reducing step). The first step is not expressible within RA since it involves *reachability* issue within a graph, whereas the second step admits a RA expression.

We then provide the sketch of the algorithm that performs the *Join* operation $\varepsilon(P \vee Q)$, where we mix for convenience procedural loop and DRC queries in Algorithm 1. In the process, θ^{ω} 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 $\varepsilon(P \vee Q)$

$\theta^{(0)} \leftarrow \{(y, y) : \exists x.M(x, y)\}$ ▷ 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 $\varepsilon(P \vee Q)$ 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 $\vee$ operation over partitions.

However, this algorithm likely results in unexpected very poor performance of ($\vee$) computation due to the gathering of a superfluous number of tuples in θ. In addition, in the last step such that $\theta^{(i+1)} = \theta^{(i)}$, the relation contains at worst all the alternative block id's assignments for the entire active domain whenever aggregation has to return a single block in the outcoming partition! It then leads to check $|\pi_2(M)|^2$ tuples and using assumption on the expected number of blocks drawn by a realistic generator (please refer to our experimental protocol in Section 4.2), we have to check $\log^2(|\pi_1(M)|)$, or basically $\log^2(|\Omega|)$ tuples. Ultimately, for every element of the active domain to be reassigned, we break the tie between all eligible solutions and choose the minimum as new

block id, that is $|\Omega| \times \log^2(|\Omega|)$ comparisons at worst, which is awful, from a computational cost.

Moreover, there is no hope of improving this computational cost by using recursive query optimizations (for instance [15]) since they only fit in cases where computation of transitive closure-like query result (such as path queries in graph) is expected. By no means, it is what we are looking for, and transitive closure is merely a *tool* but not our end purpose, that is, *transitive reduction* performing block id's reassignment.

3.4 Optimizations through SQL features

We briefly describe in this section two ways towards possible improvement in performance of query evaluation for generic set partition operations.

Early Filter Out The first optimization technique deals with the set-theoretic operators $-$ and $\cap$ performed through the membership 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 predicates $F(M\text{-block}) = F(N\text{-block})$ with $F \in \{\min, \max, \text{count}\}$ as those basic aggregate functions are efficiently implemented in R-DBMS.

Example 5. 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 make any further optimization here. Then, the refinement step would discard the second candidate pair to provide with the result set: 123.

Then we experiment a revised version of the SQL statement that takes benefits from online fast computation of aggregates (min, max, count) to filter out pairs of (M-block, N-block).

Window Functions The ANSI/ISO SQL3 introduces a nice feature called *Window Functions*, that is now part of almost all R-DBMS.This feature can be useful for the *meet* operation $P \wedge Q$ 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, we could straightforwardly assign the minimum object value of the subsets of tuples w.r.t. that partitioning to each tuple of the join.

4 Experiments

In this section, we report and discuss experimental results. The main conclusion confirms what we expected from the considerations exposed in previous Sections, *i.e.* that the SQL query engine essentially yields query processing times that are unbearably high for any non small-sized dataset.

The proposed strategies in Section 3.4 have also been implemented to fairly corroborate our claim about set partition data model mismatching within the 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 bypass the closure computation required by the join operator into the SQL framework.*

In the following, we first focus on primary SQL encoding of each operator proposed in Section 3, then we evaluate implementations of SQL tricks introduced in Section 3.4, *i.e.* pre-filtering on block id's signature for the $(-)$ operator and window function for the $\wedge$ operator.

4.1 Settings

Experiments are conducted on randomly drawn partitions. Given two partitions P and Q, we assess the performance of $P \wedge Q$, $P - Q$ and $P \vee Q$ only. Indeed, the intersection $\cap$ relies on the set difference and is formulated as a relative complement to the difference. Hence, $\wedge$, $\vee$ and $-$ are the legal baseline.

With respect to the statement 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 algorithms (Nested Loop, Index Scan, Hash Join and Merge Join) in the R-DBMS. It turns out that even if an improvement can be observed in several runs, it may also yield to a significant breakdown 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.

SQL query statements have been carefully designed to avoid usual drawbacks (useless subqueries, *etc.*) and ultimately, the execution plans have been reviewed for tracking sub-optimal evaluation strategies.

Regarding indexing strategies, we did not investigate further the regular key-based indexes since, as stated above, the main problem lies in the closure computation where indexes cannot help and second, we obviously plan to extend the atomic expressions to more complex partition queries with combination of different operators. To this end, algorithms are also required to perform well w/o auxiliary knowledge.

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.

4.2 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 $(P^{(1)}, \ldots, P^{(\ell)})$.

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 $P^{(i+1)}$.

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.

4.3 Results and analysis

Results are reported for partitions P and Q defined over $n = 1\,000$, $2,000$ and $5\,000$ items. Although those numbers are quite low, we observed in experiments that query processing times prevent 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. 3,4,5. 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 $n = 5\,000$.

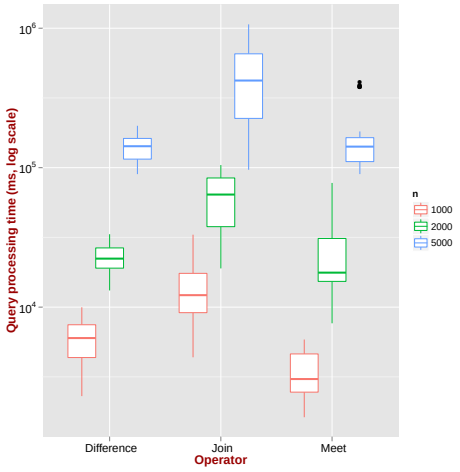

Fig. 3: Main operations $-$, $\vee$ and $\wedge$.

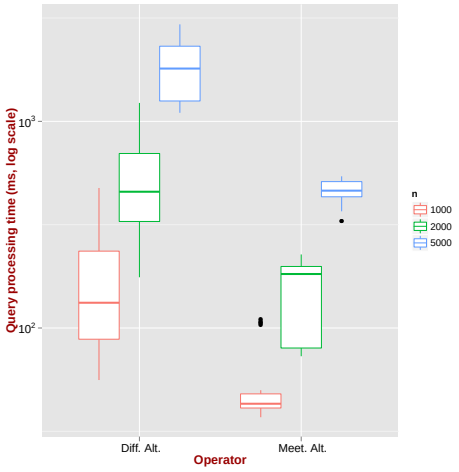

Fig. 4: *Optimized* version of ops $-$ and $\wedge$.

Regarding genuine operators, the $P \wedge Q$ operation is slightly more expensive than $P - Q$ whereas $P \vee Q$ computation performs worst than all the 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 increases the root mean square of the execution time. Ultimately, P op $P^{(\ell)}$, $\mathsf{op} \in \{\wedge \vee -\}$ 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 by 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 small to moderate size.

Moreover, though optimized versions of meet and difference operators perform much better than their respective basic implementations, it is nonetheless necessary to note that it does not outweigh the low performance of both genuine operators.

We then evaluated to which extent optimized operations could be a reliable basis to scale up to complex expression evaluation. Following this outline, Fig. 5 depicts several independent trials where we measured execution time of an increasing sequence of optimized $\wedge$, initially with only 2 operands up to 10, randomly choosen among the collection $\{P^{(0)}, P^{(1)}, \ldots, P^{(\ell)}\}$. One can observe a strict linear behavior of the execution time. It turns out that the SQL query optimizer has no impact on the behavior of such complex queries, though the collection of partitions has algebraic properties that can serve the purpose of an optimization strategy. Indeed, since partition ranks are mutually equal, it implies that the result of any meet sequence outputs a partition with a strictly lower rank. As partitions observe an increasing distance to the raw partition $P^{(0)}$, it comes that the expected result of any run is the single meet operation between the two farthest partitions of the query, *i.e.* $P^{(i)} \wedge \ldots \wedge P^{(i+k)} = P^{(i)} \wedge P^{(i+k)}$.

More generally, computing a sequence of the same lattice operator leads to finding out the equivalent query formula that combines the smallest subset of partitions to reach the lowest upper bound (resp. the highest lower bound) in the lattice. It then clearly underlies the requirement for a well-designed query plan and it is at least an open issue and a challenging task.

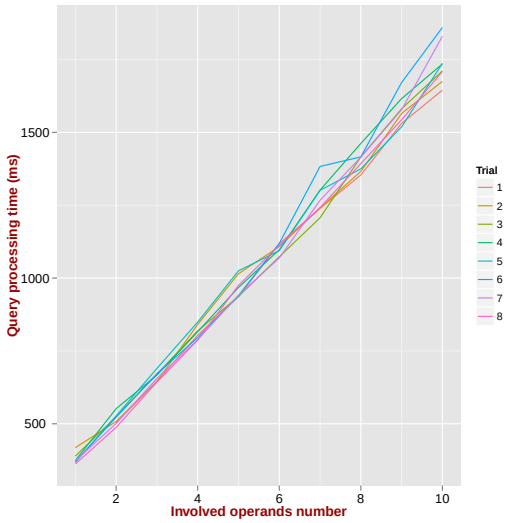

Fig. 5: SQL query processing time for increasing sequence of combination of (windowed) $\wedge$ operations with $n = 5\,000$

5 Related work

There have been efforts for supporting an algebra of sets in SQL through set-comparison queries, involving several nested queries because each set-theoretic operator requires its semantical translation in predicate calculus restricted to existential quantifiers [10]. However, query evaluation has not been assessed and it does not cover partitions as sets of sets with mutual exclusion. In [17], authors define a Minimum Description Length problem for set partition-like data structures, which 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 nevertheless focuses on the relational level built on top of the partition algebra, rather than studying partition algebra itself.

Further, lots of works, such as [2, 3, 12], have been conducted to provide 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 (trees and graphs) into relations, none of the above approaches deals with set partitions.

In another direction, there has been a 20 years line of research on *groupjoin* [18, 1, 14]. Though the merging of group-by and relational join allows for efficient computation of aggregate queries, groupjoin, whatever the implementation, has no extension to closure computation that is the critical issue of the partition join. Moreover, it can even not handle block identification by anchor since the aggregate value (min) must be associated to each and every object rather than one single representative. Roughly, the aggregate value is itself the parameter of the group-by clause.

Finally, partitions may be seen as annotated relations [11] where the annotation is essentially the label (id) of a block and the main relation represents objects. In such a context, most of the efforts have been concentrated on defining algebraic rules to support propagation of annotations along with relational operations on main tables. At the contrary, membership encoding of partitions is not intended to support any regular RA query, but rather to bring its own semantics through the ε encoding scheme and the lattice operations.

6 Conclusion

To the best of our knowledge, there has been no proposal for *generic* set partition query processing and no assessment of how computation on partitions performs, especially when the data model follows a relational-based encoding.

We provide a contribution towards achieving some relational modeling of partition through an object-block membership relation, so that it 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. We also studied its 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 information 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 optimized queries.

References

1. M. Akinde, D. Chatziantoniou, T. Johnson, and S. Kim. The MD-join: An operator for complex OLAP. In *Proc. ICDE*, pages 524–533, 2001.
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.*, 12:2335–2382, July 2011.
9. M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proc. VLDB*, pages 106–115, 1997.
10. T. Halpin and T. Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2008.
11. G. Karvounarakis and T. J. Green. Semiring-annotated data: queries and provenance? *ACM SIGMOD Record*, 3(41):5–14, 2012.
12. W. Keller. Mapping objects to tables: A pattern language. In *Proceedings of EurPLoP*, 1997.
13. A. V. Malishevski. Path independence in serial–parallel data processing. *Mathematical Social Sciences*, 27(3):335–367, 1994.
14. G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.
15. C. Ordonez. Optimization of linear recursive queries in sql. *Knowledge and Data Engineering, IEEE Transactions on*, 22(2):264–277, 2010.
16. C.-S. Park, M. H. Kim, and Y.-J. Lee. Finding an efficient rewriting of OLAP queries using materialized views in data warehouses. *Decision Support Systems*, 32(4):379–399, Mar. 2002.
17. 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.
18. G. von Bultzingsloewen. Optimizing SQL queries for parallel execution. *ACM SIGMOD Record*, 18(4):17–22, Dec. 1989.