QE, Query Enhancement

▶ Query Expansion for Information Retrieval

QoS-Based Web Services Composition

▶ Process Optimization

Quadtree Variations

▶ Quadtrees (and Family)

Quadtrees (and Family)

Michael Vassilakopoulos[1],
Theodoros tzouramanis[2]
[1]University of Central Greece, Lamia, Greece
[2]University of the Aegean, Salmos, Greece

Synonyms

Hierarchical spatial indexes; Hierarchical regular-decomposition structures; Quadtree variations

Definition

In general, the term Quadtree refers to a class of representations of geometric entities (such as points, line segments, polygons, regions) in a space of two (or more) dimensions, that recursively decompose the space containing these entities into blocks until the data in each block satisfy some condition (with respect, for example, to the block size, the number of block entities, the characteristics of the block entities, etc.).

In a more restricted sense, the term Quadtree (Octree) refers to a tree data-structure in which each internal node has four (eight) children and is used for the representation of geometric entities in a two (three) dimensional space. The root of the tree represents the whole space/region. Each child of a node represents a subregion of the subregion of its parent. The subregions of the siblings constitute a partition of the parent's regions.

Several variations of quadtrees are possible, according to the dimensionality of the space represented, the criterion guiding the subdivision of space, the type of data represented, the type of memory (internal or external) used for storing the structure, the shape, position and size of the subregions, etc. However, the term Quadtree usually refers to tree structures that divide space in a hierarchical and regular (decomposing to equal parts on each level) fashion. Since final subregions/blocks (that are no further subdivided, i.e., the blocks of the tree leaves) do not overlap, the underlying space is partitioned to a set of regions, in favor of the efficient processing of spatial queries.

Historical Background

As computer science evolved, the need for representing and manipulating spatial data (data expressing geometric properties of entities, conceptually expressed by points, line segments, regions, geometric shapes, etc.) arose in several applications areas, like Computer Graphics, Multimedia, Geographical Information Systems, or VLSI Design. The recursive decomposition of space was naturally identified as a means for organizing spatial data. The term "quadtree" was used by Finkel and Bentley [3] to express an extension of the Binary Search Tree in two dimensions that was able to index points (Point Quadtree). Since then, several Quadtree variations for a multitude of spatial data types that were used for almost all sorts of spatial-data manipulations have appeared in the literature.

The term Quadtree has taken a generic meaning and is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. The Quadtree, a variable resolution structure, is often confused with the Pyramid [11], a multi-resolution representation consisting of a hierarchy of arrays. Quadtrees and family are space-driven methods (follow the embedding space hierarchy): the division to subregions obeys a predefined way. On the contrary, R-trees and family are data-driven methods (follow the data space hierarchy): the division to subregions depends on the data.

The Region Quadtree is the most famous such structure. As its name implies, is used for representing regions and was initially termed Q-tree by Klinger and Dyer [6]. This is a main memory tree. A secondary memory implementation, the Linear Quadtree, was proposed by Gargantini [4]. The MX and PR Quadtrees are adaptations of the Region Quadtree for representing point data [11]. The MX-CIF Quadtree is a structure able to represent a collection of rectangles [11]. The PM family of Quadtrees are used for representing polygonal maps and collections of line segments. In general, extensions to three or more dimensions of each quadtree-like structure are possible. For example, the extension of the Region Quadtree for three dimensional volume data is called Region Octree. More details about the history of these and other quadtree-like structures can be found at [11,10]. There have also been proposed Quadtree variations for storing a pictorial database, like the DI-Quadtree, for binary images and the Generic Quadtree, for grayscale and color images. These and other quadtree-based methods that have been proposed for representation and querying in image database applications are reviewed in [8]. Quadtree variations have also been proposed for evolving regional data, like Overlapping Quadtrees [15], Overlapping Linear Quadtrees, the Multiversion Linear Quadtree, the Multiversion Access Structure for Evolving Raster Images and the Time-Split Linear Quadtree [13]. The XBR tree [13] is a quadtree-like structure for indexing points, or line segments especially designed for external memory. The Skip Quadtree [2] is based on Region Quadtrees and Skip Lists. It indexes points and can be used for efficiently answering point location, approximate range, and approximate nearest neighbor queries. Quadtrees have also been used in commercial Database Management Systems [7]. Demos of several Quadtree structures can be found at [1].

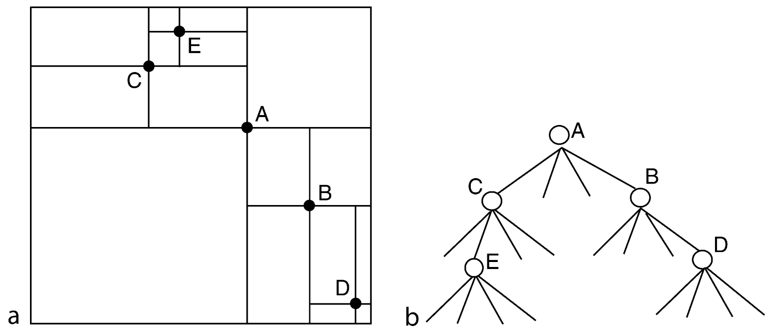
Foundations

This section briefly presents some key Quadtree structures among the numerous structures that have appeared in the literature.

Point Quadtree

The Point Quadtree [11] is an indexing mechanism for points. Each tree node corresponds to a point that subdivides the region of the node in four parts defined by two lines (parallel to the coordinate axes) on which this point lies. Thus, the shape, and position of subregions depend on the coordinates of the point (data-driven subdivision). However, each region is always subdivided in four parts. A region is considered closed in relation to its lower and left border. If multiple points with the same coordinates are allowed, each node contains a list of the points it stores. In Fig. 1, a collection of points and the resulting partitioning of space (a) and the corresponding Point Quadtree (b) are depicted. By convention, in this and other Quadtree variations, the children of a node correspond in order to the North-West, North-East, South-West and South-East subregions of the node. Note that the partitioning of space and the tree shape depend on the order of insertion of the points. Balanced versions of this tree have been proposed in the literature [11].

A Point Quadtree is not only suitable for point indexing (e.g., discovering if there is and which is the



Quadtrees (and Family). Figure 1. A collection of points (a) and the corresponding point quadtree (b).

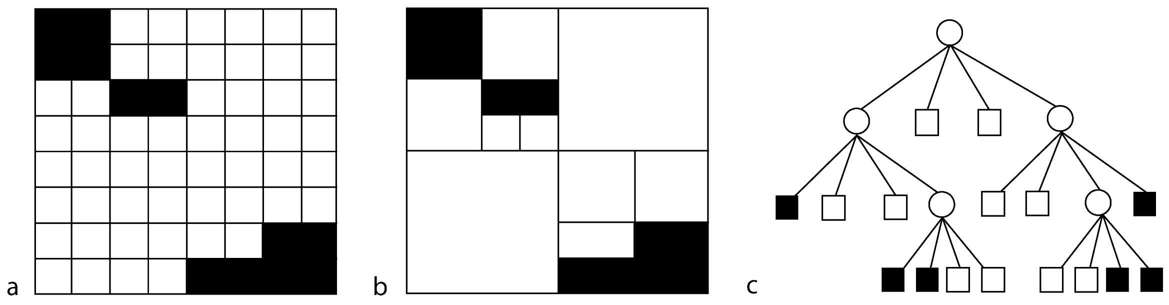
gas station that lies at a given pair of coordinates), but for answering range queries, as well (e.g., finding all the cities that reside within a specified distance from a given pair of coordinates). The efficiency of the Point Quadtree during this operation comes from pruning the search space only to nodes that may contain part of the answer.

Region Quadtree

The Region Quadtree [11] (or Q-tree [6]) is a popular hierarchical data structure for the representation of binary images, or regional data. Such an image can be represented as a $2^n \times 2^n$ binary array, for some natural number n, where an entry equal to 0 stands for a white pixel and an entry equal to 1 stands for a black pixel. The Region Quadtree for this array is made up either of a single white (black) node if every pixel of the image is white (black), or of a gray root, which points to four subquadtrees, one for every quadrant of the original image. Each region is always subdivided in four equal parts (space-driven subdivision). An example of an 8×8 binary image, its Region Quadtree and the unicolor blocks to which it is partitioned by the Quadtree external nodes are shown in Figs. 2a, 2b and 2c, respectively.

The Region Quadtree, depending on the distribution of data, may result in considerable space savings. However, it can be used for several operations on regional data. One of the them is the determination of the color of a specific pixel, or the determination of the block where this pixel resides (a specialized point-location query). Set theoretic operations are also well adapted to Region Quadtrees (recoloring, or dithering of a single image, or overlaying, union, intersection of sets of images). Connected component labeling (that is grouping pixels according to their color) is performed by utilizing Region Quadtrees. Other operations that are well adapted to Region Quadtrees include window clipping and certain transformations (like scaling by a power of two, or rotating by 90°). For more operations and details, see [9].

The pointer-based implementation of Region Quadtrees is a main memory implementation with one node type that consists of pointer fields and one color field. This node type is used for internal and external nodes according to the value of the color field. Although such an implementation is memory consuming it simplifies several operations. A secondary memory implementation of a Region Quadtree, called a Linear Quadtree [4], consists of a list of values where there is one value for each black node of the pointer-based Quadtree. The value of a node is an address describing the position and size of the corresponding block in the image. These addresses can be stored in an efficient structure for secondary memory (such as a B-tree or one of its variations). Evidently, this representation is very space efficient, although it is not suited to many useful algorithms that are designed for pointer-based Quadtrees. The most popular linear implementations are the FL (Fixed Length) and the FD (Fixed length – Depth) linear implementations. In the former implementation, the address of a black Quadtree node is a code-word that consists of n base 5 digits. Codes 0, 1, 2 and 3 denote directions NW, NE, SW and SE, respectively, while code 4 denotes a do-not-care direction. If the black node resides on level i, where $n >= i >= 0$, then the first n-i digits express the directions that constitute the path from the root to this node and the last i digits are all equal to 4. In the latter implementation, the address of a black Quadtree node has two parts: the first part is a code-word that consists of n base 4 digits. Codes 0, 1, 2 and 3 denote directions NW, NE, SW and SE, respectively. This code-word is formed in a similar way to the code-word of the FL-linear implementation with the



Quadtrees (and Family). Figure 2. A binary image (a), its partition to blocks (b) and its region quadtree (c).

difference that the last i digits are all equal to 0. The second part of the address has $[\log_2(n + 1)]$ bits and denotes the depth of the black node, or in other words, the number of digits of the first part that express the path to this node. Another interesting secondary memory implementation of the Region Quadtree is the Paged-pointer Quadtree [12] that partitions the tree nodes into pages and manages these pages using B-tree techniques.

PR Quadtree

The PR Quadtree [11] (P comes from point and R from region) is an indexing technique for points (with similar functionality to Point Quadtrees) that is based on the Region Quadtree. Points are associated with quadrants that are formed according to the Region Quadtree rules. A leaf node may be white (without any points residing in its region), or black (with one point residing in its region). In Fig. 3, the collection of points of Fig. 1 and the resulting partitioning of space (a) and the corresponding PR Quadtree (b) are depicted.

The final shape of the PR Quadtree is independent to the order of insertion of the points. A problem with this structure is that the maximum depth of recursive decomposition depends on the minimum distance between two points (if there are two points very close to each other, the decomposition can be very deep). This effect is reduced by allowing leaf nodes to hold up to C points. When this capacity is exceeded, the node is split in four. By storing leaf nodes on secondary memory and setting C according to the disk-page size, a structure that partially resides on disk is created.

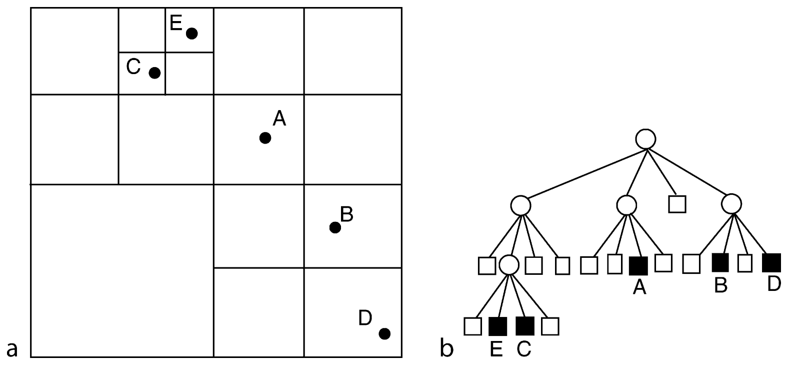
PMR Quadtree

The PMR Quadtree [11] is capable of indexing line segments and answering window queries (e.g., find the line segments that intersect a given window/area in the plane). The internal part of the tree consists of an ordinary Region Quadtree. The leaf nodes of this Quadtree are bucket nodes that hold the actual line segments. Each line segment is stored in every bucket whose quadrant (region) it crosses. A line segment can cross the region of a bucket either fully or partially. Each bucket has a maximum capacity. When this capacity is exceeded due to an insertion of a line segment, the bucket is split in four equal quadrants. However, it is possible that one (or more) of these quadrants holds a number of line segments that still exceeds the bucket capacity. Since this is not occurring very often in practical applications (e.g., when line segments represent a road network) a bucket is split only once in four and overflow buckets are created when needed. The PMR quadtree can be implemented with bucket nodes residing on disk.

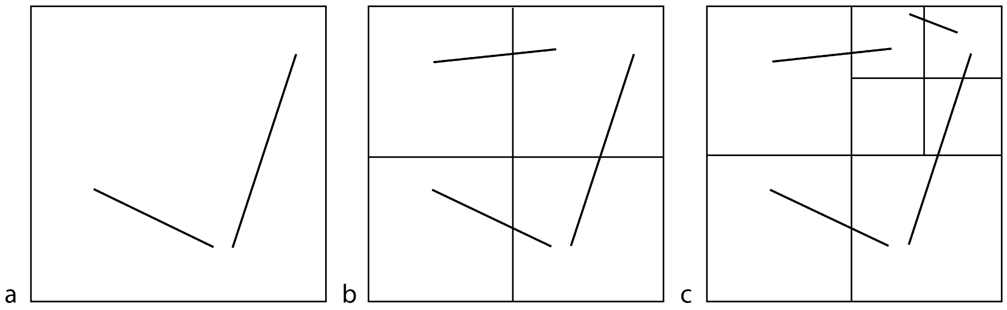
In Fig. 4, an example of the splitting of regions during the creation of a PMR Quadtree by the successive insertion of line segments is depicted. The bucket capacity is two (just for demonstration purposes). Figures 4a, 4b and 4c show the subdivision of space and the buckets created as line segments are inserted. Overflow buckets do not result from the insertions of Fig. 4. Note that the shape of the PMR Quadtree depends on the order of insertion of the line segments.

XBR Tree

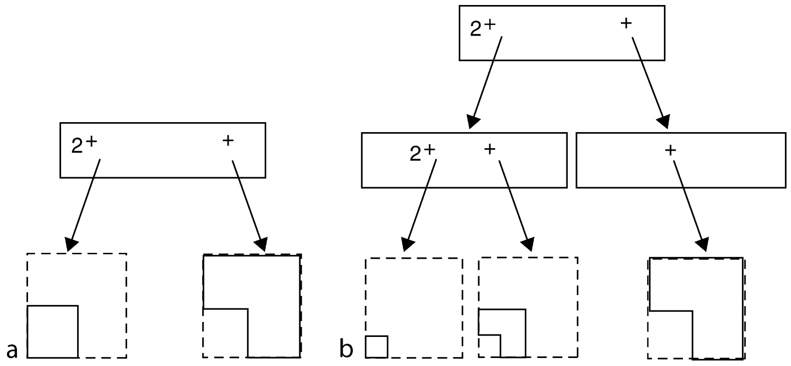
The XBR tree (XBR stands for eXternal Balanced Regular) is an indexing method suitable for indexing points (like a bucket PR Quadtree), or line segments (like a PMR Quadtree). It totally resides in secondary memory. Its hierarchical decomposition of space is the same as the one in Region Quadtrees. There are two types of nodes in an XBR-tree. The first are the internal nodes



Quadtrees (and Family). Figure 3. A collection of points (a) and the corresponding PR quadtree (b).



Quadtrees (and Family). Figure 4. Splitting of PMR-quadtree regions by the successive insertion of line segments.



Quadtrees (and Family). Figure 5. XBR trees with one level (a) and two levels (b) of internal nodes.

that constitute the index. The second are the leaves containing the data items. Both the leaves and the internal nodes correspond to disk pages.

In an internal node, a number of pairs of the form <address, pointer> are contained. The number of these pairs is non-predefined because the addresses being used are of variable size. An address expresses a child node region and is paired with the pointer to this child node. Both the size of an address and the total space occupied by all pairs within a node must not exceed the node size. The addresses in these pairs are used to represent certain subquadrants that result from the repetitive subdivision of the initial space. This is done by assigning the numbers 0, 1, 2 and 3 to NW, NE, SW and SE quadrants respectively. For example, the address 1 is used to represent the NE quadrant of the initial space, while the address 10 to represent the NW subquadrant of the NE quadrant of the initial space.

In the XBR-tree, the region of a child is the subquadrant specified by the address in its pair, minus the subquadrants corresponding to all the previous pairs of the internal node to which it belongs. Figure 5 presents XBR-trees of one (a) and two (b) levels of internal nodes. The $^+$, is used to denote the end of each variable size address. The address 2^+ in the root denotes the SW quadrant of the initial space. On the other hand, the address $^+$ in the root specifies the initial space minus the SW quadrant.

Each leaf node in the XBR tree may contain a number of data items, which is limited by a predefined capacity C. When an insertion causes the number of data items of a leaf to exceed C, the leaf is split following a hierarchical decomposition analogous to the quadtree decomposition. In case line segments are stored, (like PMR Quadtrees) a leaf is split only once in four and overflow buckets are created when needed.

Due to the incremental (level-by-level) formation of absolute addresses and the variable length coding of them, XBR trees are very compact structures. Thus, I/O during query processing is reduced, favoring processing efficiency.

Quadtree and Time-Evolving Regional Data

In Tzouramanis et al. [13] and previous papers by the same authors, four temporal extensions of the Linear Region Quadtree are presented: the Time-Split Linear

Quadtree, the Multiversion Linear Quadtree, the Multiversion Access Structure for Evolving Raster Images and Overlapping Linear Quadtrees. These methods comprise a collection of specialized quadtree-based access methods that can efficiently store and manipulate consecutive raster images. Through these methods, efficient support for spatio-temporal queries referring to the past is provided. An extensive experimental space and time performance comparison of all the above-mentioned access methods, presented in Tzouramanis et al. [13], has shown that the Overlapping Linear Quadtrees is the best performing method. For more details, see [13] and its references.

Key Applications

Quadtree-based access methods speed up access and queries in database systems that support spatial data. Some common uses of Quadtrees include representation and indexing of images, spatial indexing for several spatial types (points, regions, line segments, polygonal maps), several set operations, point location, range, and nearest neighbor queries and temporal queries on series of evolving images. Quadtrees have been employed in numerous application areas that require efficient retrieval of complex objects, such as Computer Graphics and Animation, Computer Vision, Robotics, Geographical Information Systems (GIS), Image Processing, Image and Multimedia Databases, Content-Based Image Retrieval, Medical Imaging, Urban Planning, Computer-Aided Design (CAD), or even in recent novel database applications such as P2P Networks. Furthermore, together with the R-tree family, Quadtrees serve as an important bridge for extending spatial databases to applications of several scientific areas, such as Agriculture, Oceanography, Atmospheric Physics, Geology, Astronomy, Molecular Biology, etc. Commercial database vendors like IBM and Oracle [7] have recently implemented the Quadtree and the Linear Quadtree to cater for the large and diverse above application markets.

Future Directions

Since Quadtrees were mainly introduced as main memory structures, the development of further external memory versions of several Quadtree variations and the study of their performance for several queries remain a target. Recent papers, like [5], show that the comparative performance study for several query types between space-driven and data-driven indexing techniques can lead to interesting conclusions. Algorithms for queries based on the joining of data (e.g., image data) traditionally stored in Quadtree structures and other types of spatial data stored in data-driven structures (e.g., point data stored in R-tree family structures) are worth developing and studying, especially when the evolution of the data is considered (spatio-temporal data).

Cross-references

▶ Indexing
▶ Indexing Historical Spatio-temporal Data
▶ Main Memory
▶ Multidimensional Indexing
▶ Query Processing and Optimization in Object Relational Databases
▶ R-Tree (and Family)
▶ Raster Data Management
▶ Spatial Indexing Techniques
▶ Tree-based Indexing

Recommended Reading

1. Brabec F. and Samet H. Spatial Index Demos. http://donar.umiacs.umd.edu/quadtree/index.html
2. Eppstein D., Goodrich M.T., and Sun J.Z. The skip quadtree: a simple dynamic data structure for multidimensional data. In Proc. 21st Annual Symp. on Computational Geometry, 2005, pp. 296–305.
3. Finkel R. and Bentley J.L. Quad trees: a data structure for retrieval on composite keys. Acta Informatica, 4(1):1–9, 1974.
4. Gargantini I. An effective way to represent quadtrees. Commun. ACM, 25(12):905–910, 1982.
5. Kim Y.J. and Patel J.M. Rethinking choices for multi-dimensional point indexing: making the case for the often ignored quadtree. In Proc. 3rd Biennial Conf. on Innovative Data Systems Research, 2007, pp. 281–291.
6. Klinger A. and Dyer C. Experiments on picture representation using regular decomposition. Comput. Graph. Image Process., 5:68–105, 1976.
7. Kothuri R., Ravada S., and Abugov D. Quadtree and r-tree. indexes in oracle spatial: a comparison using gis data. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 546–557.
8. Manouvrier M., Rukoz M., and Jomier G. Quadtree-Based Image Representation and Retrieval. In Spatial Databases: Technologies, Techniques and Trends. Idea Group Publishing, 2005, pp. 81–106.
9. Samet H. Applications of Spatial Data Structures. Addison Wesley, Reading, MA, USA, 1990.
10. Samet H. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.
11. Samet H. The Design and Analysis of Spatial Data Structures. Addison Wesley, 1990.
12. Shaffer C.A. and Brown P.R. A Paging Scheme for Pointer-Based Quadtrees. In Proc. 3rd Int. Symp. Advances in Spatial Databases, 1993, pp. 89–104.

13. Tzouramanis T., Vassilakopoulos M., and Manolopoulos Y. Benchmarking access methods for time-evolving regional data. Data Knowl. Eng., 49(3):243–286, 2004.
14. Vassilakopoulos M. and Manolopoulos Y. External balanced regular (x-BR) trees: New structures for very large spatial databases. In Advances in Informatics, D.I. Fotiadis, S.D. Nikolopoulos. World Scientific, 2000, pp. 324–333.
15. Vassilakopoulos M., Manolopoulos Y., and Economou K. Overlapping quadtrees for the representation of similar images. Image Vis. Comput., 11(5):257–262, 1993.

Qualitative Relations between Time Intervals

► Allen's Relations

Qualitative Temporal Constraints between Time Intervals

► Allen's Relations

Qualitative Temporal Reasoning

Paolo Terenziani
University of Turin, Turin, Italy

Synonyms

Non-metric temporal reasoning; Reasoning with qualitative temporal constraints

Definition

Qualitative temporal constraints are non-metric *temporal constraints* stating the *relative* temporal position of facts that happen in time (e.g., fact F_1 is *before* or *during* fact F_2). Different types of qualitative constraints can be defined, depending on whether facts are instantaneous, durative, and/or repeated. *Qualitative temporal reasoning* is the process of reasoning with such temporal constraints. Given a set of qualitative temporal constraints, qualitative temporal reasoning can be used for different purposes, including checking their *consistency*, determining the *strictest constraints* between pairs of facts (e.g., for query answering purposes), pointing out a *consistent scenario* (i.e., a possible instantiation of all the facts on the timeline in such a way that all temporal constraints are satisfied).

Historical Background

In several domains and\or application areas, *temporal indeterminacy* has to be coped with. In such domains, the *absolute time* when facts hold (i.e., the exact temporal location of facts) is generally unknown. On the other hand, in many of such domains, *qualitative* temporal constraints about the *relative* temporal location of facts are available, and reasoning about such constraints is an important task. As a consequence, there is a long tradition for qualitative temporal reasoning within *philosophical logic* (consider, e.g., Walker [15], who first formulated a kind of logical calculus about durative periods, and Prior's seminal branching time logics [11].

In particular, qualitative temporal reasoning is important in several artificial intelligence areas, including planning, scheduling, and natural language understanding. Therefore, starting from the beginning of the 1980s, several *specialized* approaches (as opposed to *logical* approaches, which are usually general-purpose) to the *representation* of *qualitative* temporal constraints and to temporal *reasoning* about them have been developed, especially within the artificial intelligence area.

The first milestone in the specialized approaches to qualitative temporal reasoning dates back to Allen's Interval Algebra [1], coping with qualitative temporal constraints between intervals (called *periods* by the database community, and henceforth in this entry), to cope with durative facts. Further on, several other algebrae of qualitative temporal constraints have been developed, to cope with instantaneous [14] or repeated/periodic [9,12] facts, and several temporal reasoning systems have been implemented and used in practical applications (see e.g., some comparisons in Delgrande et al. [4]).

Significant effort in the area has been devoted to the analysis of the *trade-off* between the *expressiveness* of the representation formalisms and the *complexity* of *correct* and *complete* temporal reasoning algorithms operating on them (see e.g., the survey by Van Beek [13]). Since consistency checking in Allen's algebra is NP-complete, several approaches, starting from Nebel and Burkert [10], have focused on the identification of *tractable fragments* of it. The *integration* of qualitative and metric constraints has also been analyzed (see e.g., Jonsson and Backstrom [7]). Recent developments also include *incremental* [6] and *fuzzy* [2] qualitative temporal reasoning. Moreover, starting in the 1990s, some approaches also started to investigate the adoption of qualitative temporal constraints and temporal reasoning within the temporal database context [8,3].

Foundations

Qualitative temporal constraints concern the relative temporal location of facts on the timeline. A significant and milestone example is Allen's Interval Algebra (IA). Allen pointed out the 13 primitive qualitative relations between time periods: before, after, meets, met-by, overlaps, overlapped-by, starts, started-by, during, contains, ends, ended-by, equal. These relations are *exhaustive* and *mutually exclusive*, and can be combined in order to represent disjunctive relations. For example, the constraints in (Ex.1) and (Ex.2) state that F_1 is before or during F_2, which, in turn, is before F_3.

(Ex.1) F_1 (BEFORE,DURING) F_2
(Ex.2) F_2 (BEFORE) F_3

In Allen's approach, qualitative temporal reasoning is based on two algebraic operations over relations on time periods: intersection and composition. Given two possibly disjunctive relations R_1 and R_2 between two facts F_1 and F_2, temporal intersection (henceforth $\cap$) determines the most constraining relation R between F_1 and F_2. For example, the temporal intersection between (Ex.2) and (Ex.3) is (Ex.4). On the other hand, given a relation R_1 between F_1 and F_2 and a relation R_2 between F_2 and F_3, composition (@) gives the resulting relation between F_1 and F_3. For example, (Ex.5) is the composition of (Ex.1) and (Ex.2) above.

(Ex.3) F_2 (BEFORE,MEETS,OVERLAPS) F_3
(Ex.4) F_2 (BEFORE) F_3
(Ex.5) F_1 (BEFORE) F_3

In Allen's approach, temporal reasoning is performed by a *path consistency* algorithm that basically computes the *transitive closure* of the constraints by repeatedly applying intersection and composition. Abstracting from many optimizations, such an algorithm can be schematized as follows:

```
Repeat
For all triples of facts<$F_i$,$F_k$,$F_j$>
    Let $R_{ij}$ denote the (possibly ambiguous) relation
between $F_i$ and $F_j$
    $R_{ij} \leftarrow R_{ij} \cap (R_{ik} @ R_{kj})$
    Until quiescence
```

Allen's algorithm operates in a time *cubic* in the number of periods. However, such an algorithm is *not complete* for the Interval Algebra (in fact, checking the consistency of a set of temporal constraints in the Interval Algebra is NP-hard [14]).

While in many approaches researchers chose to adopt Allen's algorithm, in other approaches they tried to design less expressive but *tractable* formalisms. For example, the Point Algebra is defined in the same way as the Interval Algebra, but the temporal elements are time points. Thus, there are only three primitive relations between time points (i.e., <,=, and>), and four disjunctive relations (i.e., (<,=), (>,=), (<,>), and (<,=,>). In the Point Algebra, sound and complete constraint propagation algorithms operate in polynomial time (namely, in $O(n^4)$, where n is the number of points [13]). Obviously, the price to be paid for tractability is expressive power: not all (disjunctive) relations between periods can be mapped onto relations between their endpoints. For instance F_1 *(BEFORE,AFTER)* F_2 cannot be mapped into a set of (possibly disjunctive) pairwise relations between time points; indeed an explicit disjunction between two different pairs of time points is needed (i.e., *(end(F_1)<start(F_2)) or (end(F_2)<start(F_1))*. The Continuous Point Algebra restricts the Point Algebra excluding inequality (i.e., (<,>)). Allen's path consistency algorithm is both sound and complete for such an algebra, and operates in $O(n^3)$ time, where n is the number of time points (for more details, see e.g., the survey by Van Beek [13]).

A different simplification of Allen's Algebra has been provided by Freksa [5]. Freska has identified coarser qualitative temporal relations than Allen's ones, based on the notion of *semi-intervals* (i.e., beginnings and ending points of durative events). As an example of relation on semi-intervals, Freksa has introduced the "older" relation (F_1 is *older* than F_2 if F_1's starting point is before F_2's starting point, with no constraint on the ending points). Notice that Freska's *older* relation corresponds to a disjunction of five Allen's relations (i.e., *before*, *meets*, *overlaps*, *finished-by*, *contains*). Freska has also shown that relations between semi-intervals result in a possible more compact notation and more efficient reasoning mechanisms, in particular if the initial knowledge is, at least in part, coarse knowledge.

Another mainstream of research about qualitative temporal reasoning focused on the identification of *tractable fragments* of Allen's algebra. The milestone work by Nebel and Burkert [10] first pointed out the "*ORD-Horn subclass*," showing that reasoning in such a class is a polynomial time problem and that it constitutes a maximal tractable subclass of Allen's algebra.

Starting in the early 1990s, some *integrated* temporal reasoning approaches were devised in order to deal with both qualitative and quantitative (i.e., metric) temporal constraints. For instance, Jonsson and Backstrom [7] proposed a framework, based on *linear programming*, that deals with both qualitative and metric constraints, and that also allows one to express constraints on the relative duration of events (see e.g., (Ex.6)).

(Ex.6) John's drive to work is at least 30 minutes more than Fred's.

Many other important issues must be taken into account when considering qualitative temporal reasoning. For example, starting from Ladkin's seminal work [9], qualitative constraints between *repeated* facts have been considered. In the same mainstream, Terenziani has proposed an extension of Allen's algebra to consider qualitative relations between periodic facts [12]. Terenziani's approach deals with constraints such as (Ex.7):

(Ex.7) Between January 1, 1999 and December 31, 1999 on the first Monday of each month, Andrea went to the post office *before* going to work.

In Terenziani's approach, temporal reasoning over such constraints is performed by a path consistency algorithm which extends Allen's one. Such an algorithm is sound but not complete and operates in cubic time with respect to the number of periodic facts.

As concerns more strictly the area of (temporal) databases, starting in the mid 1990s, some researchers started to investigate the treatment of qualitative temporal constraints within temporal (relational) databases (see e.g., [8,3]). In such approaches, the *valid time* of facts (tuples) is represented by symbols denoting time periods, and *qualitative* and *quantitative temporal constraints* are used in order to express constraints on their relative location in time, on their duration, and so on.

Koubarakis [8] first extended the constraint database model to include indefinite (or uncertain) temporal information (including qualitative temporal constraints). Koubarakis proposed an explicit representation of temporal constraints on data; moreover, the local temporal constraints on tuples are stored into a dedicated attribute. He also defined the algebraic operators, and theoretically analyzed their complexity. On the other hand, the work by Brusoni et al., [3] mainly focused on defining an integrated approach in which "standard" artificial intelligence temporal reasoning capabilities (such as the ones sketched above in this entry) are suitably extended and paired with an (extended) relational temporal model. First, the data model is extended in such a way that each temporal tuple can be associated with a set of identifiers, each one referring to a time period. A separate relation is used in order to store the qualitative (and quantitative) temporal constraints of such periods. The algebraic operations of intersection, union and difference are defined over such sets of periods, and *indeterminacy* (e.g., about the existence of the intersection between two periods) is coped with through the adoption of *conditional intervals*. Algebraic relational operators are defined on such a data model, and their complexity analyzed. Finally, an integrated and modular architecture combining a temporal reasoner with an extended temporal database is described, as well as a practical application to the management of temporal constraints in clinical protocols and guidelines.

Key Applications

Qualitative temporal constraints are pervasive in many application domains, in which the absolute and exact time when facts occur is generally unknown, while there are constraints on their relative order (or temporal location). Such domains include the "classical" domains of planning and scheduling, but also more recent ones such as managing multimedia presentations or clinical guidelines.

As a consequence, temporal reasoning is already a well-consolidated area of research, especially within the artificial intelligence community, in which a large deal of works aimed at building *application-independent* and *domain-independent* managers of temporal constraints. Such managers are intended to be *specialized knowledge servers* that represent and reason with temporal constraints, and that *co-operate* with other software modules in order to solve problems in different applications. For instance, in planning problems, a temporal manager could co-operate with a planner, in order to check incrementally the temporal consistency of the plan being built. In general, the adoption of a specialized temporal manager is advantageous from the computational point of view (e.g., with respect to general logical approaches based on theorem proving), and it allows programmers to focus on their domain-specific and application-specific problems and to design modular architectures for their systems.

On the other hand, the impact and potentiality of extensively exploiting qualitative temporal reasoning in temporal databases have only been minimally explored by the database community, possibly due to the computational complexity that it necessarily involves. However, (temporal) databases will be increasingly applied to new applications domains, in which the structure and the inter-dependencies of facts (including the temporal dependencies) play a major role, while the assumption that the absolute temporal location of facts is known does no longer hold. Significant application areas include database applications to store workflows, protocols, guidelines (see, e.g., the example in [3]), and so on. To cope with such applications, "hybrid" approaches in which qualitative (and/or quantitative) temporal reasoning mechanisms are paired with classical temporal database frameworks (e.g., along the lines suggested in [3]) are likely to play a significant role in a near future. The role of qualitative temporal constraints (and temporal reasoning) may be even more relevant at the *conceptual* level. Several temporal extensions to conceptual formalisms (such as the Entity-Relationship) have been proposed in recent years, and there is an increasing awareness that, in many domains, qualitative (and/or quantitative) temporal constraints between conceptual objects are an intrinsic part of the conceptual model. As a consequence, qualitative temporal reasoning techniques such as the ones discussed above, may in the near future, play a relevant role at the conceptual modeling level.

Future Directions

One of several possible future research directions of qualitative temporal reasoning, which may be particularly interesting for the database community, is its application to "Active Conceptual Modeling." In his keynote talk at ER'2007, Prof. P. Chen, the creator of the Entity-Relationship model, has stressed the importance of extending traditional conceptual modeling to "Active Conceptual Modeling." Roughly speaking, the term "active" denotes the need for coping with evolving models having learning and prediction capabilities. Such an extension is needed in order to adequately cope with a new range of phenomena, including disaster prevention and management. Chen has stressed that "Active Conceptual Modeling" requires, besides the others, an explicit treatment of time. The extension and integration of qualitative temporal reasoning techniques into the "Active Conceptual Modeling" context is likely to give a major contribution to the achievement of predictive and learning capabilities, and to become a potentially fruitful line of research.

Cross-references

- ▶ Absolute Time
- ▶ Allen's Relations
- ▶ Relative Time
- ▶ Temporal Constraints
- ▶ Temporal Indeterminacy
- ▶ Temporal Integrity Constraints
- ▶ Temporal Periodicity
- ▶ Time in Philosophical Logic
- ▶ Time Period
- ▶ Valid Time

Recommended Reading

1. Allen J.F. Maintaining knowledge about temporal intervals. Commun. ACM, 26(11):832–843, 1983.
2. Badaloni S. and Giacomin M. The algebra IA^{fuz}: a framework for qualitative fuzzy temporal reasoning. Artif. Intell., 170 (10):872–908, 2006.
3. Brusoni V., Console L., Pernici B., and Terenziani P. Qualitative and quantitative temporal constraints and relational databases: theory, architecture, and applications. IEEE Trans. Knowl. Data Eng., 11(6):948–968, 1999.
4. Delgrande J., Gupta A., and Van Allen T. A comparison of point-based approaches to qualitative temporal reasoning. Artif. Intell., 131(1–2):135–170, 2001.
5. Freksa C. Temporal reasoning based on semi-intervals. Artif. Intell., 54(1–2):199–227, 1992.
6. Gerevini A. Incremental qualitative temporal reasoning: algorithms for the point algebra and the ORD-Horn class. Artif. Intell., 166(1–2):37–80, 2005.

7. Jonsson P. and Backstrom C. A unifying approach to temporal constraint reasoning. Artif. Intell., 102:143–155, 1998.
8. Koubarakis M., Database models for infinite and indefinite temporal information. Inf. Syst., 19(2):141–173, 1994.
9. Ladkin P. Time representation: a taxonomy of interval relations. In Proc. 5th National Conf. on AI, 1986. pp. 360–366.
10. Nebel B. and Burkert H.J. Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra. J. ACM, 42(1):43–66, 1995.
11. Prior A.N. Past, Present and Future. Oxford University Press, Oxford, 1967.
12. Terenziani P. Integrating calendar-dates and qualitative temporal constraints in the treatment of periodic events. IEEE Trans. Knowl. Data Eng., 9(5):763–783, 1997.
13. Van Beek P. Reasoning about qualitative temporal information. Artif. Intell., 58(1–3):297–326, 1992.
14. Vilain M. and Kautz H. Constraint propagation algorithms for temporal reasoning. In Proc. 5th National Conf. on AI, 1986, pp. 377–382.
15. Walker A.G. Durèes et instants. La Revue Scientifique, (3266), p. 131, 1947.

Quality and Trust of Information Content and Credentialing

Chintan Patel, Chunhua Weng
Columbia University, New York, NY, USA

Definition

The quality and reliability of biomedical information is critical for practitioners (clinicians and biomedical scientists) to make important decisions about patient conditions and to draw key scientific conclusions towards developing new drugs, therapies and procedures. Evaluating the quality and trustworthiness of biomedical information [1] requires answering questions such as, *where did the data come from*, *under what conditions was the data generated, how accurate and complete is the data,* and so on.

Key Points

The quality and reliability of biomedical information is dependent on the task or the context of the application. There is a basic set of domain-independent features that can be used to characterize the quality and trustworthiness of the information:

Accuracy: The correctness of the information. Inherent noisiness in the underlying data generating clinical processes or biological experiments often leads to various errors in the resulting data. It is critical to quantify the frequency and source of such errors using the measure of accuracy to enable the clinicians and researchers for making informed decisions using the data.

Completeness: In biomedical settings, it is often necessary to have all the pertinent (complete) information at hand while making important clinical decisions or performing complex biological experiments. There are various constraints due to limited technology and the nature of clinical practice that leads to incomplete biomedical data:

1. In healthcare settings, only partial data gets recorded in electronic form and vast amount of data is still only available on paper [4]. For example, various clinical notes such as admit, progress and discharges notes containing valuable clinical information are generally written on paper charts and not entered electronically.
2. In several instances, despite the availability of electronic data, the information is not usable or accessible due to differences in underlying data standards, information models, terminology, etc. Consider for example, two biological databases using different ontologies for protein annotation, which will not be able to support data reuse or sharing across.

Transparency: An important aspect for determining the trustworthiness of information is the understanding or knowledge of the underlying processes or devices generating the data. Various tools or resources that provide transparency to data sources are more likely to be trusted by clinicians or biologists [2].

Credentialing: The trustworthiness of information in turn depends on the level of trust in data originators. Hence, it is important to attribute the information to its source in order to allow consumers of information to make appropriate decisions. Consider for example that a biological annotation reviewed by human curators will be trusted more than an annotation automatically extracted from literature using text-mining [3].

As the biomedical domain becomes more and more information driven, the methods and techniques to characterize the quality and trustworthiness of information will gain more prominence.

Cross-references

► Clinical Data Quality and Validation

Recommended Reading

1. Black N. High-quality clinical databases: breaking down barriers. Lancet, 353(9160):1205–1211, 1999.
2. Buza T.J., McCarthy F.M., Wang N., Bridges S.M., and Burgess S.C. Gene Ontology annotation quality analysis in model eukaryotes. Nucl. Acids Res., 36(2):e12, 2008.
3. D'Ascenzo M.D., Collmer A., and Martin G.B. PeerGAD: a peer-review-based and community-centric web application for viewing and annotating prokaryotic genome sequences. Nucl. Acids Res., 32(10):3124–3159, 2004.
4. Thiru K., Hassey A., and Sullivan F. Systematic review of scope and quality of electronic patient record data in primary care. BMJ, 26(7398):1070, 2003.

Quality Assessment

► Clustering Validity

Quality of Data Warehouses

Rafael Romero[1], Jose-Norberto Mazón[1], Juan Trujillo[1], Manuel Serrano[2], Mario Piattini[2]
[1]University of Alicante, Alicante, Spain
[2]University of Castilla – La Mancha, Spain

Definition

Quality is an abstract and subjective aspect for which there is no universal definition. It is usually said that there is a quality definition for each person. Perhaps the most abstract definition for this topic is that the data warehouse quality means the data is suitable for the intended application by all users. In this way, it is very complex to measure or assess the quality of a data warehouse system. Normally, the *data warehouse* quality is determined by: (i) the quality of the data presentation, and (ii) the quality of the *data warehouse* itself. The latter is determined by the quality of the Database Management System (DBMS), the data quality and the quality of the underlying data models used to design it. A good design may (or may not) lead to a good data warehouse, but a bad design will surely render a bad data warehouse of low quality. In order to measure the quality of a data warehouse, a key issue is defining and validating a set of measures to help to assess the quality of a data warehouse in an objective way, thus guaranteeing the success of designing a good data warehouse.

Historical Background

Few works have been presented in the area of objective indicators or measures for data warehouses. Instead, most of the current proposals for data warehouses still delegate the quality of the models to the experience of the designer.

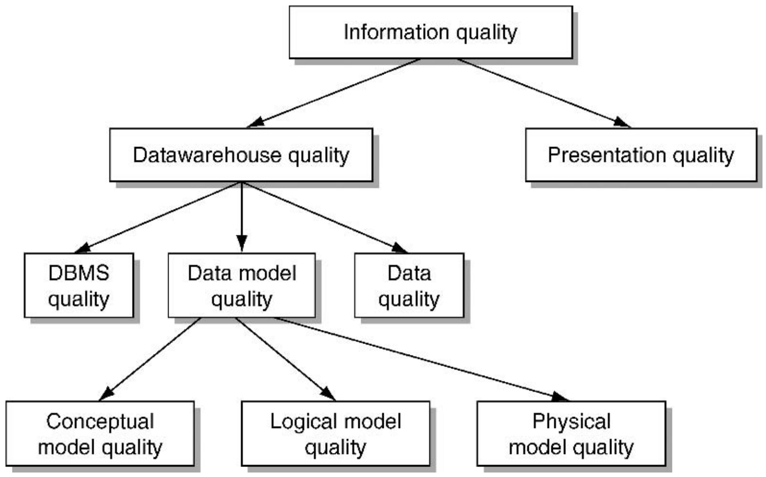
Only the model proposed by Jarke et al. [5], which is described in more depth in Vassiladis' Ph.D. thesis [14], explicitly considers the quality of conceptual models for data warehouses. Nevertheless, these approaches only consider quality as intuitive notions. In this way, it is difficult to guarantee the quality of data warehouse conceptual models, a problem which has initially been addressed by Jeusfeld et al. [6] in the context of the DWQ (Data Warehouse Quality) project. This line of research addresses the definition of measures that allows designers to replace the intuitive notions of quality of conceptual models of the data warehouse with formal and quantitative measures. Sample research in this direction includes normal forms for data warehouse design as originally proposed in [8] and generalized in [7]. These normal forms represent a first step towards objective quality metrics for conceptual schemata.

Following the idea of assessing the quality of data warehouses in an objective way, several measures for evaluating the quality of data warehouse logical models have been proposed in recent years and validated both formally and empirically [11,12].

Lately, Si-Saïd and Prat [13] have proposed some measures for multidimensional schemas analyzability and simplicity. Nevertheless, none of the measures proposed so far has been empirically validated, and therefore, their practical utility has not been proven.

Foundations

The information quality of a data warehouse is determined by: (i) the quality of the system itself, and (ii) the quality of the data presentation (see Fig. 1). In fact, it is important that the data of the data warehouse not only correctly reflects the real world, but also that the data are correctly interpreted. Regarding data warehouse quality, three aspects must be considered: the quality of the DBMS (Database Management System) that supports it, the quality of the data models



Quality of Data Warehouses. Figure 1. Quality of the information and the data warehouse.

used in their design (conceptual, logical and physical), and the quality of the data contained in the data warehouse. The presentation quality of data warehouses is more related to the data presentation according to front-end tools such as OLAP (On-Line Analytical Processing), data reporting or data mining. For this reason, the following issues pertaining to data warehouse quality are described next: (i) DBMS quality, (ii) data model quality, and (iii) data quality.

Quality of DBMS

The quality of the DBMS in which the data warehouse is implemented is important, since the database engine is the core of the data warehouse system and has a deep impact on the performance of the whole system. A good DBMS could improve the performance of the system and the quality of the data by implementing constraints and integrity rules. In order to assess the quality of a DBMS, several international standards can be used, such as ISO/IEC 9126 [3] and ISO/IEC 9075 [4] or even information from database benchmarks [9].

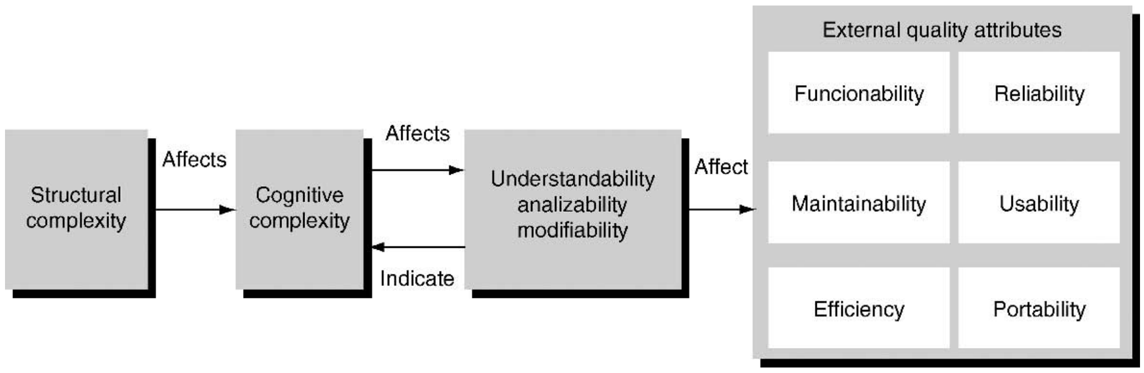
Quality of Data Warehouse Data Models

The quality of the data models used in the design of data warehouses relies on the quality of the conceptual, logical, and physical models used for its design. A first step towards obtaining high quality data models is the definition of development methodologies. However, a methodology, though necessary, may not be sufficient to guarantee the quality of a data warehouse. Indeed, a good methodology may (or may not) lead to a good product, but a bad methodology will surely render a bad product of low quality. Furthermore, many other factors could influence the quality of the products, such as human decisions. Therefore, it is necessary to complete specific methodologies with measures and techniques for product quality assessment.

Structural properties (such as structural complexity) of a software artifact have an impact on its cognitive complexity as shown in Fig. 2. Cognitive complexity means the mental burden on those who have to deal with the artifact (e.g., developers, testers, maintainers). High cognitive complexity of an artifact reduces its understandability and leads to undesirable external quality attributes as defined in the standard ISO/IEC 9126 [3]. The model presented in Fig. 2 is an adaptation of the general model for software artifacts proposed in Briand et al. [2].

Indeed, as data warehouse models are software artifacts, it is reasonable to consider that they follow the same pattern. It is thus important to investigate the potential relationships that can exist between the structural properties of these schemas and their quality factors.

In order to get a valid set of data warehouse measures, the definition of measures should be based on clear measurement goals and the measures should be defined following the organization's needs related to external quality attributes. In defining measures, it is also advisable to take into account the expert's



Quality of Data Warehouses. Figure 2. Relationship between Structural Properties, Cognitive Complexity, Understandability and External Quality Attributes – based on the work described in Briand et al. [12].

knowledge. Figure 3 presents a method (based on the method followed in [11,12]) for obtaining valid and useful measures. In this figure, continuous lines show measure flow and dotted lines show information flow.

This method has five main phases starting at the identification of goals and hypotheses and leading to the measure application, accreditation, and retirement:

1. *Identification*: Goals of the measures are defined and hypotheses are formulated. All of the subsequent phases will be based upon these goals and hypotheses.

2. *Creation*: This is the main phase, in which measures are defined and validated. This phase is divided into three sub phases:

a) *Measures definition*. Measure definition is made by taking into account the specific characteristics of the system to be measured, the experience of the designers of these systems and the work hypotheses. A goal-oriented approach as GQM (Goal-Question-Metric [1]) can also be very useful in this step.

b) *Theoretical validation*. The formal (or theoretical) validation helps identify when and how to apply the metrics. There are two main tendencies in measuring formal validation: the frameworks based on axiomatic approaches [7] and the ones based on measurement theory [10]. The goal of the former is merely definitional, as in this type of formal framework, a set of formal properties is defined for given software attributes and it is possible to use this set of properties for classifying the proposed measures. On the other hand, in the frameworks based on measurement theory, the information obtained is the scale to which a measure pertains and, based on this information, statistics and transformations, which can be applied to the measure, can be known.

c) *Empirical validation*. The goal of this step is to prove the practical utility of the proposed measure. Empirical validation is crucial for the success of any software measurement project as it helps to confirm and understand the implications of the measurement of products. Although there are various ways perform to this step, the empirical validation can be divided into experiments, case studies, and surveys.

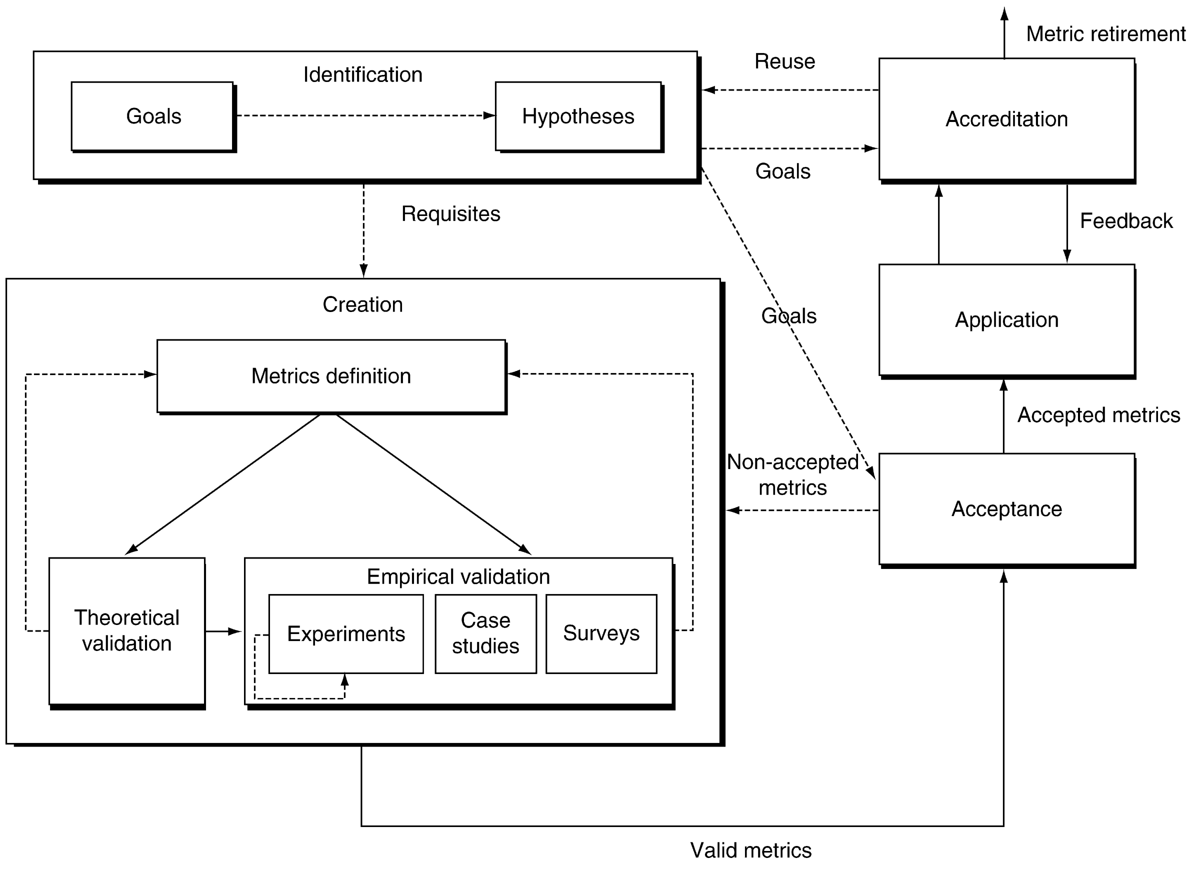
This process is evolutionary and iterative and as a result of the feedback, the measure could be redefined or discarded, depending on their formal and empirical validation. As a result of this phase a valid metric is obtained.

3. *Acceptance*: The goal of this phase is the systematic experimentation of the measure. This is applied in a context suitable to reproduce the characteristics of the application environment, with real business cases and real users, to verify its performance against the initial goals and stated requirements.

4. *Application*: The accepted measure is used in real cases.

5. *Accreditation*: This is the final phase of the process. It is a dynamic phase that proceeds simultaneously with the application phase. The goal of this phase is the maintenance of the measure, so it can be adapted to the changing environment of the application. As a result of this phase, the measure can be retired or reused for a new measure definition process.

The most important data models for measuring quality are conceptual and logical models as the quality of physical models has to do with performance issues and the physical distribution of data. Following the above method, a set of measures can be defined and validated for data warehouse conceptual models



Quality of Data Warehouses. Figure 3. Metrics creation process.

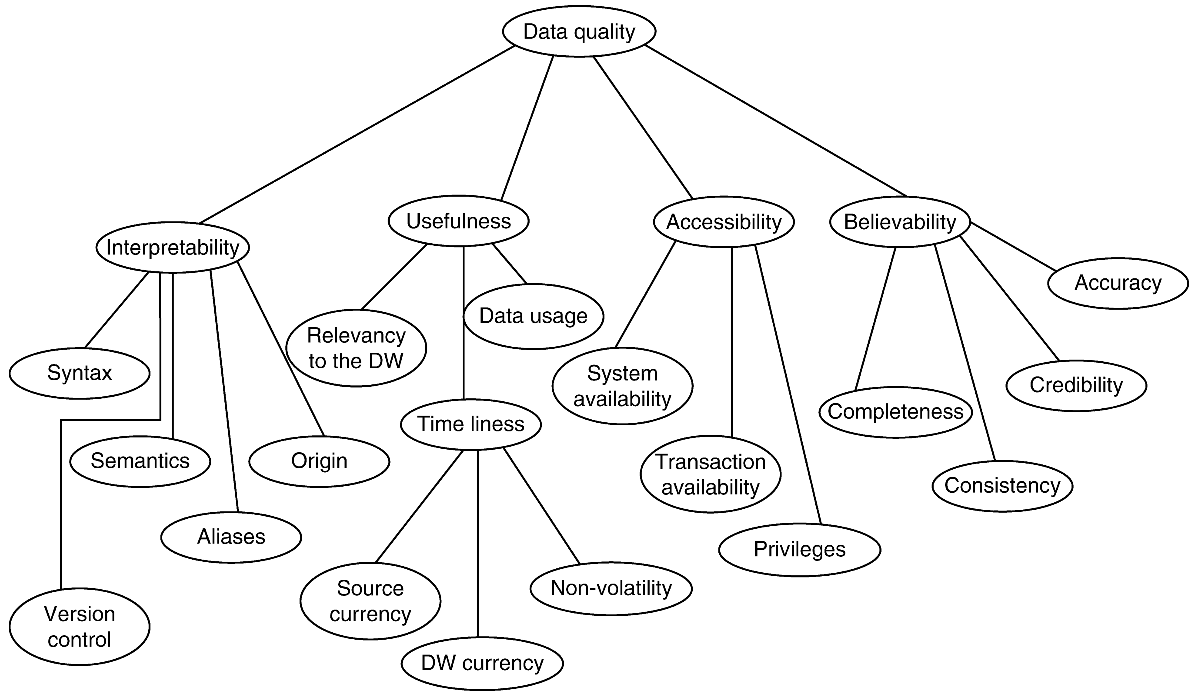
quality [12]. Some examples of the measures defined in [12] are: number of dimensions of the multidimensional conceptual model, number of hierarchy levels of the multidimensional conceptual model, ratio of hierarchy levels (i.e., number of hierarchy levels per dimension of the multidimensional conceptual model), and the maximum depth of the hierarchy relationships of the multidimensional conceptual model.

Taking into account these characteristics, a set of measures for data warehouses logical models have also been defined and validated in [11]. Specifically, these measures are defined for a relational implementation (tables, columns, foreign keys, etc.). Some examples of the measures defined in [11] are: number of fact tables of the schema, number of shared dimension tables (i.e., number of dimension tables shared for more than one star (fact) of the schema), number of foreign keys in all the fact tables of the schema, and ratio of schema attributes (i.e., number of attributes in dimension tables per attributes in fact tables).

Data Quality

Finally, data quality is a multidimensional concept that is made up of several aspects, all of which depend on the needs of the data consumers or system designers. The most accurate definition for data quality is fitness for use, i.e., how suitable the data is for a concrete task. This concept can be defined in terms of particular criteria, related either to the data life-cycle or to the way data are meant to be used. Usually, the way of determining the data quality of an application is to define a framework that specifies the data quality dimensions in a concrete domain. The reader is referred to the DWQ project [6]. Figure 4 shows the quality factors as stated in the DWQ project.

As stated in the DWQ project, data warehouse data quality is related to several characteristics or quality dimensions, such as interpretability, usefulness, accessibility and believability. These quality dimensions should be interpreted as the ability to understand, use, access and trust the data that is stored in



Quality of Data Warehouses. Figure 4. Data warehouse quality factors.

the data warehouse. Each dimension tackles a different aspect of data quality and is broken up into several sub-categories, such as timeliness, availability, clarity, and accuracy.

When dealing with interpretability, the data must be easy to understand, and special attention should be placed on the format and structure of the data. In this sense, the syntax of the data should be inspected, since aliases and abbreviations may be eluded and the origin and the different versions of the data must be controlled.

Data relevancy and timeliness are the main problems to be tackled in data usefulness. Data relevancy can be analyzed in ETL processes, while timeliness is more related to the currency of the data. It is universally known that data warehouses are typically not updated everyday, but irrelevant data cannot be allowed in the system. A good schedule of data warehouse refresh process can avoid this problem.

Accessibility is related to system availability and access privileges. Data warehouse systems should be in stable and well-dimensioned systems and privileges should be studied in great detail. It is also necessary to test and maintain the system periodically.

Believability is a very important aspect of data warehouse data quality, as it deals with completeness, consistency, credibility and accuracy of the data. Probably this quality dimension is the most studied by data quality experts, but it is usually the most difficult to assess. This quality dimension is not only related to source data quality and ETL processes' quality, it is also related to the data warehouse update processes that can pollute the data warehouse data. In this way, data cleaning algorithms could be very useful.

In order to detect data quality problems, it is advisable to have indicators that can help identify the different quality threats for each of the data warehouse data quality dimensions. There are several quality indicator proposals, but they are not oriented to the data warehouse field and are only usually oriented to a specific quality dimension such as accuracy, completeness and timeliness. Further research has to be done in this field in order to obtain a complete set of data warehouse data quality indicators.

Key Applications

The main application of Data Warehouse Quality is oriented to the work of data warehouse designers. Introducing quality aspects from early phases of data warehouse design is a good way to improve the quality and success of a data warehouse. In this way, quality models and quality driven design methods are good approaches to design successful data warehouse systems. These design methods should be based on the metrics and techniques that have been discussed in the previous sections.

Future Directions

Some open research lines within data warehouse quality are: (i) to provide a complete set of quality measures that allows to measure and assess data warehouse quality models in different dimensions such as the complexity, understandability, usability and so on (Fig. 2), (ii) the traceability of the measures throughout all the design steps (conceptual, logical and physical), and (iii) the measure's thresholds, which should be the last step in defining measures, i.e., being able to provide a number for each measure under which one can completely assure the quality of a model. The latter is definitely the most difficult issue that has not yet been covered.

Experimental Results

Generally speaking, before doing an experiment with data quality measures, the settings must first be defined (including the main goal of the experiment, the subjects that will participate in the experiment, the main hypothesis under which the experiment will be run, the independent and dependent variables to be used in the model, the experimental design, the experiment running, material used and the subjects that performed the experiment). After that, the collected data validation must be discussed. Finally, the results are analyzed and interpreted to find out if they follow the formulated hypothesis. Only after a complete set of experiments, and following the method presented in Fig. 3, a measure can be accepted or rejected.

The reader is referred to the work of the Software Engineering Lab [15] for expanding the concepts about experimental process. A complete set of experimental works can be found in [11,12]. In those works several data warehouse quality measures are empirically validated as data warehouse quality indicators.

Cross-references

▶ Data Warehouse
▶ Data Warehouse Life-Cycle and Design
▶ Data Warehouse Security

Recommended Reading

1. Basili V. and Weiss D.A. Methodology for collecting valid software engineering data. IEEE Trans. Software Eng., 10:728–738, 1984.
2. Briand L., Morasca S., and Basili V. Property-based software engineering measurement. IEEE Trans. Software Eng., 22 (1):68–86, 1996.
3. ISO/IEC, 9126: Software Engineering – Product quality. Geneva, Switzerland, 2003.
4. ISO/IEC, 9075: Information Technology – Database languages. Geneva, Switzerland, 2006.
5. Jarke M., Lenzerini M., Vassiliou Y., and Vassiliadis P. Fundamentals of Data Warehouses. Springer, 2002.
6. Jeusfeld M.A., Quix C., and Jarke M. Design and analysis of quality information for data warehouses. In Proc. 17th Int. Conf. on Conceptual Modeling, 1998, pp. 349–362.
7. Lechtenbörger J. and Vossen G. Multidimensional normal forms for data warehouse design. Inf. Syst., 28:415–434, 2003.
8. Lehner W., Albretch J., and Wedekind H. Normal forms for multidimensional databases. In Proc. 10th Int. Conf. on Scientific and Statistical Database Management, 1998, pp. 63–72.
9. Othayoth R. and Poess M. The Making of TPC-DS. In Proc. 32nd Int. Conf. on Very Large Data Bases, 2006, pp. 1049–1058.
10. Poels G. and Dedene G. DISTANCE: a framework for software measure construction, Research Report DTEW9937, Department of Applied Economics, Katholieke Universiteit Leuven, Belgium, 1999, p. 46.
11. Serrano M., Calero C., and Piattini M. Validating metrics for data warehouses. IEE Proc. Software, 149(5):161–166, 2002.
12. Serrano M., Trujillo J., Calero C., and Piattini M. Metrics for data warehouse conceptual models understandability. Inf. Software Technol., 49(8):851–870, 2007.
13. Si-Saïd S. and Prat N. Multidimensional schemas quality: assessing and balancing analyzability and simplicity. In Proceedings of the ER 2003 Workshops, 2003, pp. 140–151.
14. Vassiliadis P. Data Warehouse Modeling and Quality Issues, Ph.D. Thesis, National Technical University of Athens, Athens, Greece, 2000.
15. Wohlin C., Runeson P., Höst M., Ohlson M., Regnell B., and Wesslén A. Experimentation in Software Engineering: An Introduction. Dordrecht, Kluwer Academic, 2000.

Quantiles on Streams

Chiranjeeb Buragohain[1], Subhash Suri[2]
[1]Amazon.com, Seattle, WA, USA
[2]University of California-Santa Barbara, Santa Barbara, CA, USA

Synonyms

Median; Histogram; Selection; Order statistics

Definition

Quantiles are order statistics of data: the ϕ-quantile $(0 \leq \phi \leq 1)$ of a set S is an element x such that $\phi|S|$ elements of S are less than or equal to x and the remaining $(1 - \phi)|S|$ are greater than x. This entry describes data stream (single-pass) algorithms for computing an approximation of such quantiles.

Historical Background

Since the earliest days of data processing, there has been a need to summarize data. Large volumes of raw, unstructured data easily overwhelm the human ability to comprehend or digest. Tools that help identify the major underlying trends or patterns in data have enormous value. Quantiles characterize distributions of real world data sets in ways that are less sensitive to outliers than simpler alternatives such as the mean and the variance. Consequently, quantiles are of interest to both database implementers and users: for instance, they are a fundamental tool for query optimization, splitting of data in parallel database systems, and statistical data analysis.

Quantiles are closely related to the familiar concepts of frequency distributions and histograms. The cumulative frequency distribution $F()$ is commonly used to summarize the distribution of a (totally ordered) set S. Specifically, for any value x,

$$F(x) = \text{Number of values less than } x. \qquad (1)$$

The quantile $Q(\phi)$, or the ϕ-th quantile is simply the inverse of $F(x)$. Specifically, if the set S has n elements, then the element x has the property that

$$Q(F(x)/n) = x. \qquad (2)$$

Thus, the 1⁄2-quantile is just the familiar median of a set, while 0- and 1-quantiles are the minimum and the maximum elements of the set. Histograms are another popular method for summarizing data into a smaller number of "buckets": the buckets only retain the information how many elements fall between two consecutive bucket boundaries, but not their precise values. It is easy to see that a sequence of quantiles resembles a *histogram* of the underlying set, and provides a natural and complete summary of the entire *distribution* of the data values.

In computer science, sorting and selection (another name for quantile computation) are two of the most basic problems, with long and intellectually rich history of research. Indeed, the computational complexity of selection, namely, determining the element of a given rank, is one of the earliest celebrated problems, and the elegant, linear-time algorithm of Blum et al. [2] is a classical result, taught regularly in the undergraduate algorithms and data structures course. For more recent theoretical results on the complexity of selection in the classical comparison-complexity model, please refer to the survey by Paterson [16].

Foundations

The problem with quantile computation, while well-solved in the classical model of computation, assumes a new and challenging character within the constraints of single-pass computation (the streaming model). Indeed, when the algorithm's memory is limited and significantly smaller than the size of the data set S, it is not possible to compute the quantile precisely, and the best possible solution is an approximation. This was formalized in a 1980 paper by Munro and Paterson [15], who proved that any algorithm that determines the median of a set by making at most p sequential scans of the input requires at least $\Omega(n^{1/p})$ working memory. Thus, computing the true median will require memory linear in the size of the set.

Against this backdrop, the main focus of recent research has been on achieving provable-quality approximation of the quantiles. In particular, an ε-approximate quantile summary of a sequence of n elements is a data structure that can answer quantile queries about the sequence to within a precision of εn. In other words, when queried for a ϕ-quantile, for $0 \leq \phi \leq 1$, the structure returns an element x that is guaranteed to be in the $[\phi - \varepsilon, \phi + \varepsilon]$ quantile range. The key evaluation metric for the performance of these approximation schemes is the size (memory footprint) of their summary data structures, although other factors such as simplicity of implementation are also desirable.

The discussion in this entry will focus on algorithms that operate in the *data stream* model: the algorithm is endowed with a finite memory, which is significantly smaller in size than the size of the input. The input is presented to the algorithm in an arbitrary (perhaps adversarial) order, and any data not explicitly stored by the algorithm is irretrievably lost. Thus, the algorithm is restricted to a single scan of the data in the input order, and after this scan it must output an approximation of the quantiles of the input values.

Before discussing the state of the art for this problem, it may help to consider a real-world scenario for the use of quantiles in data streams, both to illustrate a motivating application and to appreciate the scale of the problem. A web site, such as a search engine, consists of several web server hosts. The users' queries (requests) are collectively handled by these servers (using some scheduling protocol); and the overall performance of the web site is characterized by the latency (delay) encountered by the users. The distribution of the latency values is typically very skewed, and a common practice is to track some particular quantiles, for instance, the 95th percentile latency. In this context, one can ask several questions.

- What is the 95th percentile latency of a single web server?
- What is the 95th percentile latency of the entire web site (over all the servers)?
- What is the 95th percentile latency of the website during the last 1 h?

The Yahoo website, for instance, handles more than 3.4 billion hits per day, which translates to 40,000 requests per second. The Wikipedia website handles 30,000 requests per second at peak, with 350 web servers.

While all three questions relate to computing of quantiles, they have different technical nuances, and often require different algorithmic approaches. In particular, the first question is the most basic version, asking for a determination of quantiles for a stream of data; the second extends the setting to *distributed input*, and thus demands an algorithm in the distributed computing model. The third problem is an instance of the *sliding window* model, where the computation must occur over a subset (time window) of the stream, and this subset is continuously changing. This entry is primarily focused on the stream setting (problem 1), also known as the cash register model, but will also discuss, when appropriate, extensions to these other models.

Randomized Algorithms

One can estimate the quantiles of a stream by the quantiles of a random sample of the input. The key idea is to maintain a random sample of appropriate size and when asked for a quantile of the input set, simply report the corresponding quantile of the random sample. If the size of the input stream, N is known, then the following simple algorithm can compute a random sample of size k in one-pass: choose each element independently with probability k/N to include in the sample. If the size of the full data stream is not known in advance, or if the ability to answer queries during reading the stream is required as well, then the reservoir sampling algorithm of Vitter [19] can be used instead. To maintain a sample of size k, the reservoir sampling algorithms begins by including the first k stream elements in the sample; from then on, the ith element from the stream is chosen with probability i/n. If the ith element is chosen, one of the elements from the current sample is evicted uniformly at random to keep the size of the sample constant at k. While straightforward to implement, random sampling has the disadvantage of needing a rather large sample to achieve expected approximation accuracy. Specifically, in order to estimate the quantiles with precision εn, with probability at least $1 - \delta$, a sample of size $\Theta(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ is required, where $0 < \delta < 1$.

In [3], Cormode and Muthukrishnan proposed a more space-efficient data structure, called Count-Min sketch, which is inspired by Bloom filters. Count-Min sketch allows ε-approximation of quantiles using $O(\frac{1}{\epsilon} \log^2 n \log(\frac{\log n}{\phi\delta}))$ memory. Although the space needed by Count-Min is worse than the two deterministic schemes discussed below, it has the advantage of allowing general updates to the streams: past elements can be deleted as well as their values updated.

Deterministic Algorithms

The first deterministic streaming algorithm for quantiles was proposed by Manku et al. [13,14], building on the prior work by Munro and Paterson [2]. This algorithm has space complexity $O(\frac{1}{\epsilon} \log^2 \epsilon n)$, meaning that using memory that grows poly-logarithmically in the stream size and inversely with the accuracy parameter ε, the quantiles can be estimated with precision εn. This result has since been improved by two groups: in [10], Greenwald and Khanna propose a $O(\frac{1}{\epsilon} \mathit{log}\ \epsilon n)$ memory scheme, and in [18], Shrivastava et al. propose a $O(\frac{1}{\epsilon} \log U)$ memory scheme, where U is the size of domain from which the input is drawn.

The Greenwald-Khanna (GK) algorithm is based on the idea that if a sorted subset $\{v_1, v_2, \ldots\}$ of the input stream S (of current size n) can be maintained such that the ranks of v_i and v_{i+1} are within 2ε of each other, then an arbitrary quantile query can be answered with precision εn. Their main contribution is to show how to maintain such a subset of values using a data structure of size $O(\frac{1}{\epsilon} \log \epsilon n)$. The Q-Digest scheme of Shrivastava et al. [18] approaches the quantile problem as a histogram problem over a universe of size U; thus $\log U$ is the number of bits needed to represent each element. Q-Digest maintains a set of buckets dynamically, merging those that are light (containing few items of the stream) and splitting those that are heavy, with an aim to keep the relative sizes of all the buckets nearly equal. Specifically, using a $O(\frac{1}{\epsilon}\log U)$ size data structure, Q-Digest ensures that the input stream is divided into $O(1/\epsilon)$ buckets, with each bucket containing $O(\epsilon n)$ items. Thus, the rank of any item can be determined with precision εn by locating its bucket.

In theoretical terms, the GK scheme has better performance when the input is drawn from a large universe, but the stream itself has only modest size. The Q-Digest, on the other hand, is superior when the stream size is huge but elements are drawn from a smaller universe. The GK algorithm is very clever, but requires a sophisticated analysis. The Q-Digest is simpler to understand, analyze, and implement, and it lends itself to easy extensions to distributed settings.

Practical Considerations

A detailed study of the empirical performance of quantile algorithms was carried out by Cormode et al. [5] on IP stream datasets. They concluded that with careful implementation, a commodity hardware machine (dual Pentium 2.8 GHz CPU and 4 GB RAM) can keep up with a 2 Gbit/s stream (310,000 packets per second). Performance numbers can depend also on the input distribution. For example, the deterministic algorithms presented above can have different memory usage and accuracy depending on the order in which the input values are presented, but sketching techniques such as the Count-Min sketch are not affected by the order of the input. The input value distribution can also impact perceived accuracy of the approximate quantiles. For example, for skewed distributions, the *numeric value* of the exact ϕ-th quantile can be arbitrarily far from the numeric values of the $(\phi \pm \epsilon)$-th quantile.

Extensions

Given the fundamental nature of quantiles and their widespread applications in data processing, it is no surprise that there are multiple extensions of the basic setting that have been considered so far. There are many interesting and practically-motivated applications, such as the latency of the web site mentioned earlier, where quantiles must be computed over distributed data, or over a sliding window portion of the stream etc. In the following, the current state of algorithms for these variants are briefly discussed.

Quantiles in Distributed Streams In many settings, data of interest are naturally distributed across multiple sources, such as servers in a web application and measurement devices in a sensor network. In these applications, it is necessary to compute the quantile summary of the entire data, but *without* creating a centralized repository of data, which could be undesirable because of the additional latency, communication overhead, or energy constraints of untethered sensors. The efficiency of an algorithm in this distributed setting is measured by the amount of information each node in the system must transmit during the computation.

One natural approach for distributed approximation of quantiles is for each node (server, sensor, etc.) to compute a local summary of its data, and arrange the nodes in a virtual hierarchy that guides them to merge these summaries into a final structure computed at the root of the hierarchy. The tree-based Q-digest [18] algorithm extends rather easily to the distributed setting, as the histogram boundaries of the Q-Digest are aligned to binary partition of the original value space U. The space complexity of the distributed version remains the same as the stream version, namely, $O(\frac{1}{\epsilon}\log U)$. The GK algorithm is little more complicated to extend to distributed streams, but Greenwald and Khanna themselves developed such an extension in [9]. However, the space complexity of their distributed data structure grows to $O(\frac{1}{\epsilon}\log^3 n)$ [9]. The Bloom filter based Count-Min sketch [3] also extends easily to the distributed setting also without any increase in the space complexity.

Quantiles in Sliding Windows In many applications, the user is primarily interested in the most recent portion of the data stream. This poses the *sliding window* extension of the quantiles problem, in which the desired quantile summary for the most recent N data elements – the window slides with the arrival of each new element, as the oldest element of the window is discarded and the new arrival added. In [12], Lin et al. presented such a sliding window scheme for quantile summaries, however, the space requirement for their algorithm is $O(\frac{1}{\epsilon^2} + \frac{1}{\epsilon}\log \epsilon^2 N)$. This was soon improved by Arasu and Manku [1] to $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\log N)$.

Biased Estimate of Quantiles The absolute measure of approximation precision is quite reasonable as long as the error εn is quite small compared to the rank of the quantile sought, namely, ϕn. This holds as long as the quantiles of interest are in the middle of the distribution. But if ϕ is either close to 0 or 1, one may prefer a *relative* error, so that the estimated quantile is in the range $[(1-\varepsilon)\phi, (1+\varepsilon)\phi]$. This variant was solved by Gupta and Zane [13] using random sampling techniques with a $O(\frac{1}{\epsilon^3}\log n \log\frac{1}{\delta})$ size data structure. The space bound has since been improved by Cormode et al. [4] to $O(\frac{1}{\epsilon}\log U \log(\epsilon n))$ using a deterministic algorithm.

Duplicate Insensitive Quantiles In some distributed settings, a single event can be observed multiple times. For example in the Internet, a single packet is observed at multiple routers. In wireless sensor networks, due to the broadcast nature of the medium, and to add fault-tolerance, data can be routed along multiple paths. Summaries such as quantiles or the number of distinct items are clearly not robust against duplication of data; on the other hand, simpler statistics such as minimum and maximum are not affected by duplication. Flajolet and Martin's distinct counting algorithm [8] is the seminal work in this direction. Cormode et al. have introduced algorithms based on sampling to compute various duplicate insensitive aggregates [6]. Their Count-Min sketch can be also easily adapted to compute duplicate insensitive quantiles.

Key Applications

Internet-scale network monitoring and database query optimization are two important applications that originally motivated the need for quantiles summaries over data streams. Gigascope [7] is a streaming database system that employs statistical summaries such as quantiles for monitoring network applications and systems. Quantile estimates are also widely used in query optimizers to estimate the size of intermediate results, and use those estimates to choose the best execution plan [5]. Distributed quantiles have been used to succinctly summarize the distribution of values occurring over a sensor network [8]. In a similar context, distributed quantiles are also used to summarize performance of websites and distributed applications [17].

Future Directions

The field of computing approximate quantiles over streams have led to a fertile research program and is expected to bring up new challenges in both theory and implementation. Although there is an obvious lower bound of $\Omega(\frac{1}{\epsilon})$ memory required to compute ε-approximate quantiles, there is no known non-trivial lower bound on memory. Since the current best algorithms requires $O(\frac{1}{\epsilon}\log(\epsilon n))$ or $O(\frac{1}{\epsilon}\log(U))$ memory, it will be useful to either lower the memory usage or prove a better lower bound.

Another direction in which improvements are highly desirable is running time. The current deterministic algorithms require amortized running time of $O(\log\frac{1}{\epsilon} + \log\log(\epsilon n))$ or $O(\log\frac{1}{\epsilon} + \log\log(U))$ per item. In high data-rate streams, even such low processing times are not fast enough: what is desired is a $O(1)$ insert time, or even a sublinear time quantile algorithm. As of now, there is no memory efficient sub-linear time quantile algorithm known except for random sampling.

Cross-references

- ▶ Adaptive Stream Processing
- ▶ Approximate Query Processing
- ▶ Continuous Query
- ▶ Data Aggregation in Sensor Networks
- ▶ Data Stream
- ▶ Distributed Data Streams
- ▶ Distributed Query Processing
- ▶ Geometric Stream Mining
- ▶ Hierarchical Heavy Hitter Mining on Streams
- ▶ In-Network Query Processing
- ▶ Stream Mining
- ▶ Stream Processing
- ▶ Streaming Applications

Recommended Reading

1. Arasu A. and Manku G.S. Approximate counts and quantiles over sliding windows. In Proc. 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2004, pp. 286–296.
2. Blum M., Floyd R., Pratt V., Rivest R., and Tarjan R.E. Time bounds for selection. J. Comput. Syst. Sci., 7:448–461, 1973.
3. Cormode G. and Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. J. Algorithms, 55(1):58–75, 2005.
4. Cormode G., Korn F., Muthukrishnan S., and Srivastava D. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In Proc. 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2006, pp. 263–272.
5. Cormode G., Korn F., Muthukrishnan S., Johnson T., Spatscheck O., and Srivastava D. Holistic UDAFs at streaming speeds. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 35–46.

6. Cormode G., Muthukrishnan S., and Zhuang W. What's different: distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 57.
7. Cranor C., Johnson T., Spataschek O., and Shkapenyuk V. Gigascope: a stream database for network applications. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 647–651.
8. Flajolet P. and Martin G.N. Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci., 31(2):182–209, 1985.
9. Greenwald J.M. and Khanna S. Power-conserving computation of order-statistics over sensor networks. In Proc. 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2004, pp. 275–285.
10. Greenwald J.M. and Khanna S. Space-efficient online computation of quantile summaries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001, pp. 58–66.
11. Gupta A. and Zane F. Counting inversions in streams. In Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms, 2003, pp. 253–254.
12. Lin X., Lu H., Xu J., and Yu J.X. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In Proc. 20th Int. Conf. on Data Engineering, 2004, pp. 362–374.
13. Manku G.S., Rajagopalan S., and Lindsay B.G. Random sampling techniques for space efficient online computation of order statistics of large datasets. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1999, pp. 251–262.
14. Manku G.S., Rajagopalan S., and Lindsay B.G. Approximate medians and other quantiles in one pass and with limited memory. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998, pp. 426–435.
15. Munro J.I. and Paterson M.S. Selection and sorting with limited storage. Theor. Comput. Sci., 12:315–323, 1980.
16. Paterson M.S., Progrees in Selection. In Proc. Scandinavian Workshop on Algortithm Theory, 1996, pp. 368–379.
17. Pike R., Dorward S., Griesemer R., and Quinlan S. Interpreting the data: parallel analysis with sawzall. Sci. Program. J., 13 (4):227–298, 2005.
18. Shrivastava N., Buragohain C., Agrawal D., and Suri S. Medians and beyond: new aggregation techniques for sensor networks. In Proc. 2nd Int. Conf. on Embedded Networked Sensor Systems, 2004, pp. 239–249.
19. Vitter J.S. Random sampling with a reservoir. ACM Trans. Math. Softw., 11(1):37–57, 1985.

Quantitative Association Rules

Xingquan Zhu
Florida Atlantic University, Boca Ration, FL, USA

Synonyms

Numeric association rules

Definition

Quantitative association rules refer to a special type of association rules in the form of $X \rightarrow Y$, with X and Y consisting of a set of numerical and/or categorical attributes. Different from general association rules where both the left-hand and the right-hand sides of the rule should be categorical (nominal or discrete) attributes, at least one attribute of the quantitative association rule (left or right) must involve a numerical attribute. Examples of this type of association rule can be categorized into the following two classes, depending on whether the rules are measured by the frequency of the supporting data records or by some distributional features of some numerical attributes.

1. *Frequent Rules:* Out of all applicants whose age is between 30 and 39 and marriage status is "yes," ($\rightarrow$) 95% of them have two cars, and 10% applicants in the database satisfy this rule.
2. *Distributional Rules:* If the patient is non-smoker and wine-drinker then ($\rightarrow$) his/her average life expectancy is 85 (whereas the average life expectancy in the overall population is 80).

In the above first example, "applicant age" is a numerical attribute, and "marriage status" and "# of cars" are categorical attributes. The support and confidence measures indicate that out of all the data records matching the left-hand of the rule, 95% of them satisfy the whole rule (*Confidence*) and there are 10% of the data records which hold this rule (Support). The key to discovering this type of rules is to discretize numerical attributes into discrete regions, so the frequency of the data records satisfying the rule can be measured and the general Apriori-based association rule mining approaches [1] can apply.

Distributional rules denote a set of quantitative association rules with instances covered by the rules rendering themselves statistically different from the overall population (with respect to some statistical measures, such as mean and standard deviation values). Because the calculation of the statistical values over numerical attributes is straightforward, attribute discretization is unnecessary, which consequently avoids possible information loss from the data discretization. The key to discovering this type of rules is to find a small group of instances (with respect to the left-hand side of the rules) statistically and significantly different from the overall population, if the right-hand of the rules is considered.

Historical Background

The problem of discovering quantitative association rule was first introduced by Srikant and Agrawal [4], and the main focus then was to discover frequent rules. The solution proposed in the paper [4] adopts a data discretization for mining association rules. Their approach first partitions the values of the numerical attributes into fine intervals and then combines adjacent intervals if necessary. A partial completeness measure was introduced to quantify the information loss due to the partitioning, so the algorithm can properly determine the number of partitions. Traditionally, numerical attribute discretization can be achieved through the following two approaches, with the latter preferred in practice:

- *Equal-width discretization* divides the range of a numerical attribute into N intervals of equal width. For example, ages from 20 to 60 can be divided into four intervals of 10-year width. This method can be easily implemented, but subject to a clear drawback that there may be too few instances in some intervals and too many in other intervals, and both cases hinder mining high quality association rules.
- *Equal-depth discretization* divides the range of numerical attribute into N intervals such that each interval equally contains $1/N$ of the total instances. This method avoids the possible imbalance inherent in the equal-width discretization method, but it may separate similar attribute values into different intervals and group dissimilar attributes into the same interval.

Noticing the adoption of the information discretization can lead to unavoidable information loss or misleading association rules, Aumann and Lindell [2] suggested distributional quantitative association rules, where the general structure of the rules take the form of:

$$\text{population} - \text{subset} \rightarrow \text{interesting} - \text{behavior}$$

More specifically, the left-hand of the rule confines a population subset, whereas the right-hand of the rule provides the statistical measures of the confined subset. The objective is to discover subsets significantly different from the whole population (with respect to the underlying measures, e.g., mean or standard deviation values). Following this principle, Webb [5] extended the framework to include other measures such as the minimum and the count measures.

Foundations

Frequent Rules

To demonstrate the procedures of the frequent quantitative association rule discovery, take the toy database in Fig. 1(a) as an example [4], where attribute "Age" is numerical, and "Married" and "NumCars" are categorical attributes. Assuming the user specified *Support* and *Confidence* values are 40% and 50% respectively, it means that a prospective rule (the left-hand and the right-hand together) should cover at least two records ($5 \times 40\%$). For all data records satisfying the left-hand of the rule, 50% of them should also contains the right-hand sides of the rule ($X \rightarrow Y$). The major steps of discovering frequent quantitative association rules can then be summarized as follows [4]:

1. Determining the partition numbers and the region of partitioning for each numerical attribute. E.g., Fig. 1(a) lists four partitions for "Age" (denoted by [20,24], [25,29], [30,34], and [35,39]), with each region mapping to one integer value {1,2,3,4}.
2. Applying each mapping table to all records of the corresponding numerical attribute, with numerical values replaced by the matching integer values. The example of the mapped database is shown in Fig. 1(c).
3. Generating frequent itemsets based on the mapped database and the user specified *Support* value (the existing Apriori like association rule mining methods can be applied directly).
4. Using discovered frequent itemsets to generate quantitative association rules, with each frequent itemset decomposed into two (left- and right-hand) components. For example, if itemset "ABCD" is found frequent, a possible quantitative association rule can be made by decomposing "ABCD" as "AB" → "CD." As long as the validate check asserts that confidence value of this rule ("AB" → "CD") is greater than the user specified value (*Confidence*), the rule is taken as a valid rule.
5. Collect all quantitative association rules generated from the above process and prune redundant rules. *E.g.*, if "AB" → "CD" and "AB" → "CDE" are both valid rules, "AB" → "CD" can be pruned as it can be generalized (inferred) from "AB" → "CDE".

People

RecordID	Age	Married	Numcars
100	23	No	1
200	25	Yes	1
300	29	No	0
400	34	Yes	2
500	38	Yes	2

a

Mapping age

Interval	Integer
20..24	1
25..29	2
30..34	3
35..39	4

b

After mapping attributes

RecordID	Age	Married	Numcars
100	1	2	0
200	2	1	1
300	2	2	1
400	3	1	2
500	4	1	2

c

Frequent itemsets: sample

Itemsets	Support
{ ⟨Age: 20..29⟩ }	3
{ ⟨Age: 30..39⟩ }	2
{ ⟨Married: Yes⟩ }	3
{ ⟨Married: No⟩ }	2
{ ⟨Numcars: 0..1⟩ }	3
{ ⟨Age: 30..39⟩, ⟨Married: Yes⟩ }	2

d

Rules: sample

Rule	Support	Confidence
⟨Age: 30..39⟩ and ⟨Married: Yes⟩ ⇒ ⟨Numcars: 2⟩	40%	100%
⟨Age: 20..29⟩ ⇒ ⟨Numcars: 0.1⟩	60%	66.6%

e

Quantitative Association Rules. Figure 1. Example of problem decomposition for quantitative association rule mining (revised from [2]).

Distributional Rules

Different from the frequent quantitative association rules, where the main challenge is to determine the "optimum" number of partitions and the region of partitioning, the distributional rules represent a set of quantitative association rules, where the statistical features of the samples covered by the rule are different from the whole population. Similar to general association rules, the distributional rules also contain the left- and the right-hand. The left-hand side of the rule is a description of a subset of the database, while the right-hand side provides a description of outstanding behaviors of this subset. A rule is only interesting if the mean for the subset (specified by the left-hand side) is significantly different from the rest and is therefore unexpected. Consider the following distributional rule, the left-hand side consists of two categorical attributes (Non-smoker: {Yes or No}, and Wine-drinker: {Yes or No}), and the right-hand side is a numerical attribute (life expectancy). The rule is considered informative and meaningful as it indicates that individuals characterized by the left-hand side of the rule (Non-smoker and wine-drinker) have a longer average life expectancy (85) than the whole population (80).

$$\begin{aligned}&\text{Non} - \text{smoker and wine} - \text{drinker}\\ &\quad \rightarrow \text{life expectancy} = 85\ (\text{overall} = 80)\end{aligned}$$

Following this definition, one can easily extend the framework to allow one or multiple numerical attributes to appear on the left- or right-hand sides of the rule (or both), or employ other statistical measures rather than the mean values to assess the rules.

In order to discover distributional rules, Aumann and Lindell [3] proposed two methods to discover the following two types of rules:

- $X \rightarrow Mean_J\ (T_X)$, where X and J denote a single numerical attribute, T_X denotes transactions confined by attribute X and $\mathrm{Mean_J}(T_X)$ represents the mean value of attribute J (for all samples in T_X).
- $X \rightarrow M_J\ (T_X)$, where X denotes one or multiple categorical attributes, J consists of one or multiple quantitative attributes, and M means one particular statistical measure (there is no restriction on the number of attributes in X and J).

The solution to the first type of distributional rules is straightforward, since the rules only involve two numerical attributes (one on each side), an algorithm can afford to go through each pair of numerical attributes to discover meaningful rules. More specifically, for any two numerical attributes *i* and *j*, one can first sort all records in the database based on the attribute *i*, then any above or below average continuous region of values in *j* can form a prospective quantitative rule. For example, given the toy database in Fig. 2(a) which records the age and the size of the striped bass, the sorted database with respect to attribute *i* (age) is given in Fig. 2(a). The average of attribute *j* for the top three records (001, 003, and 002) is 0.77, which is significantly lower than the mean of the whole population (1.83). Therefore, the below average region (001, 003, and 002) forms a meaningful quantitative association rule denoted by:

$$\text{Age} \leq 2 \rightarrow \text{Mean weight } 0.77 \text{ lbs (Overall } 1.83)$$

Because any above or below average continuous region of values in *j* can form a prospective quantitative rule, one can continuously span the region with respect to the attribute *j*, and discover the maximum region satisfying the user specified requirements (e.g., α times less/larger than the average).

In order to discover the second type of distributional quantitative association rules, one can employ a two-stage approach, which applies general association rule to the whole database by considering categorical attributes only, followed by a refining process to check each rule by considering the numerical attribute values [5].

1. *Discovering frequent itemsets:* Finding all frequent itemsets by considering categorical attributes of the database only (this can be easily achieved through existing Apriori-like algorithms [1].
2. *Calculating statistical distribution values:* For each numerical attribute, calculate the value of the distribution measures (mean/variance) over samples confined by each frequent itemset. For example, if "Non-smoker and Wine-drinker" are found frequent (i.e.,, a frequent itemset), all samples matching this itemset form a sample set *P*, from which the statistical distribution value of a numerical attribute can be calculated.
3. *Refining quantitative association rules:* For every frequent itemset (denoted by *X*) and one numerical attribute *e*, the algorithm continuously check if $X \rightarrow Mean_e\ (T_X)$ and $X \rightarrow Variance_e\ (T_X)$ are meaningful rule (comparing to the whole population). In addition, for any two rules $X \rightarrow Mean_e\ (T_X)$ and $Y \rightarrow Mean_e\ (T_Y)$, the algorithm will check whether the former is a sub-rule of the latter, or vice versa, such that the algorithm can output compact rules with minimum redundancy.

Key Applications

Business intelligence, market basket analysis, fraud detection (fraud medical insurance claims).

Future Directions

All of the above techniques intend to discover quantitative association rules in the forms of the conjunction of individual attributes. One interesting problem is to find quantitative association rules with (linearly or nonlinearly) combined attributes. For example, finding rules like "Age $\leq \alpha \rightarrow$ Length/Weight $\leq \beta$." Here the right-hand of the rule is a non-linear combination of the numerical attributes (Length and Height), and α and β are some discovered values. In [3], *Ruckert* et al. proposed a quantitative association rule mining approach which is able to discover similar rules with linearly weighted attributes like "Age $\leq \alpha \rightarrow \alpha_1$·Length + α_2·Weight $\leq \beta$." Future research may emphasize the generalized quantitative association

ID	*i* (Age)	*j* weight (lbs)
001	1.0	0.5
002	2.0	0.8
003	1.5	1.0
004	2.5	1.7
005	4.0	3.8
006	3.5	3.2
Mean	2.42	1.83

a Original table

ID	*i* (Sorted age)	*j* weight (lbs)
001	1.0	0.5
003	1.5	1.0
002	2.0	0.8
004	2.5	1.7
006	3.5	3.2
005	4.0	3.8
Mean	2.42	1.83

0.77

b Sorted by attribute *i*

Quantitative Association Rules. Figure 2. Example of distributional association rule mining (age and weight of the striped bass).

rule discovery, where rules consist of non-linearly combined attributes.

Another interesting problem concerning quantitative association rules is to discover relational patterns of the quantitative association rules across multiple databases. *E.g.*, Finding patterns that are frequent with a support level of α in database A, but significantly infrequent with a support level of β in databases B or/and C. In [6], Zhu and Wu proposed a hybrid frequent pattern tree based solution to address this problem with a focus on the general association rules. Extending the problem of relational frequent pattern discovery to quantitative association rules is another interesting topic for future research.

Cross-references

▶ Association Rules

Recommended Reading

1. Agrawal R., Imielinski T., and Swami A. Mining association rules between sets of items in large databases. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1993, pp. 207–216.
2. Aumann Y. and Lindell Y. A statistical theory for quantitative association rules. In Proc. 5th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 1999, pp. 261–270.
3. Ruckert U., Richter L., and Kramer S. Quantitative association rules based on half-spaces: an optimization approach. In Proc. 2004 IEEE Int. Conf. on Data Mining, 2004, pp. 507–510.
4. Srikant R. and Agrawal R. Mining quantitative association rules in large relational tables. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1996, pp. 1–12.
5. Webb G.I. Discovering associations with numeric variables. In Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2001, pp. 383–388.
6. Zhu X. and Wu X. Discovering relational patterns across multiple databases. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 726–735.

QUEL

Tore Risch
Uppsala University, Uppsala, Sweden

Definition

QUEL was the query language used in the original Ingres system from Berkeley University.

Key Points

QUEL was one of the first relational database query languages. It can be seen as a syntactically sugared tuple relational calculus language. The Postgres extension of Ingres originally used an extention of QUEL called PostQUEL, but was later replaced with SQL.

Cross-references

▶ Ingres
▶ Postgres
▶ Tuple Relational Calculus

Query Answering in Analytical Domains

▶ Query Processing in Data Warehouses

Query Assistance

▶ Web Search Query Rewriting

Query by Example

▶ Video Querying

Query by Humming

Yingyi Bu[1], Raymond Chi-Wing Wong[2],
Ada Wai-Chee Fu[1]
[1]Chinese University of Hong Kong, Hong Kong, China
[2]Hong Kong University of Science and Technology, Hong Kong, China

Synonyms

Music retrieval; Time series database querying

Definition

With the appearance of large scale audio and video databases in various application areas, novel information retrieval methods adapted to the specific characteristics of these data types are required. A natural way of searching in a musical audio database is by humming the tune of a song as a query, which is so-called "query by humming". In this entry, state-of-the-art techniques for effective and efficient querying by humming are described.

Historical Background

In 1995, Asif Ghias et al. [1] proposed the basic architecture for a system supporting query by humming. Three main components are introduced in the system: a pitch-tracking module, a melody database, and a query engine. Queries are hummed into a microphone, digitized, and fed into a pitch-tracking module. Then, a symbol sequence representation upon the relative pitch transitions of the hummed melody is sent to the query engine, which produces a ranked list of matching melodies.

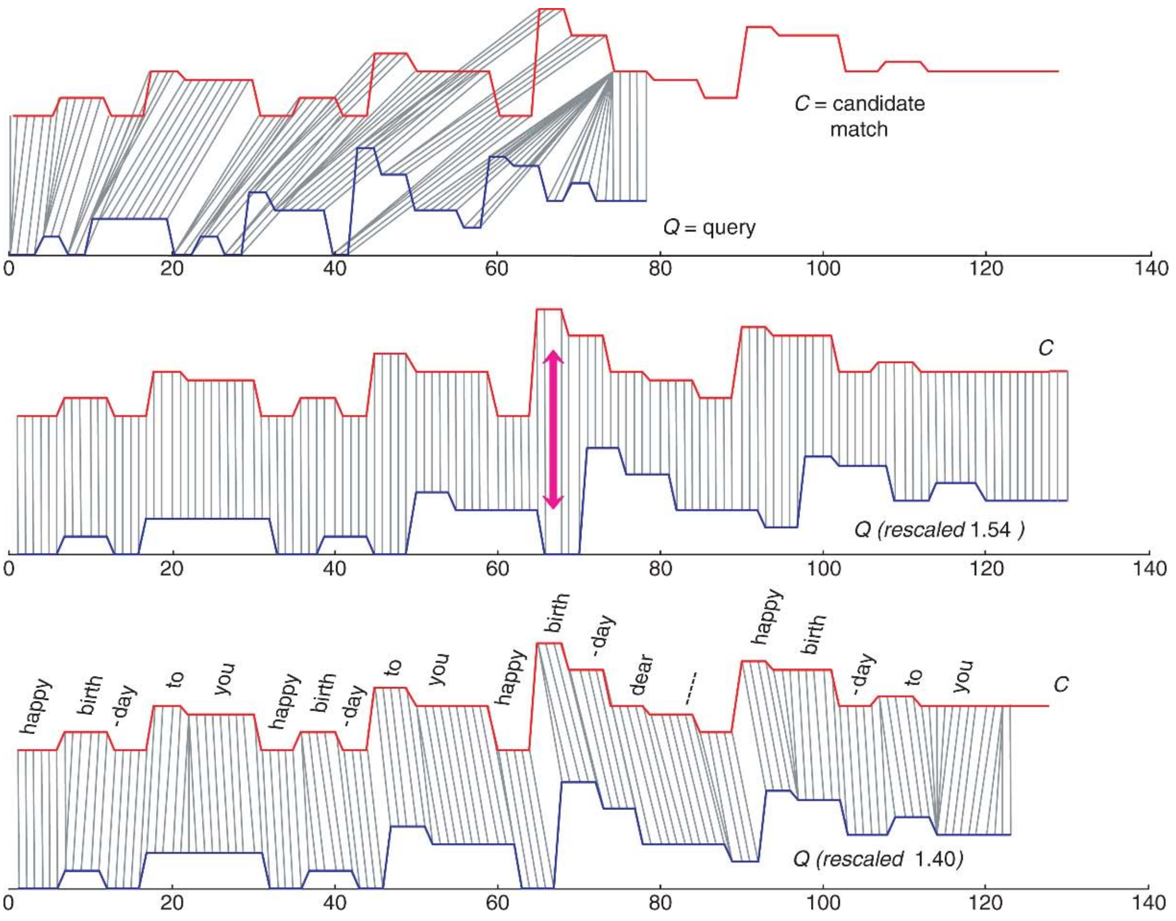
Foundations

In recent researches, musical data are modeled as time series which are real valued sequences rather than symbol sequences. In speech comparisons, small fluctuation of the tempo of the speaker could be allowed in order to identify similar contents. There have been some works that match a melody more effectively by considering warping and scaling in humming queries. This generally gives better query results because it is free from the error-prone note segmentation. However, those works rely on distance measures such as universal scaling (US), dynamic time warping (DTW) [2,4] and scaling and time warping (SWM) [5], the efficiency might be rather poor. Fortunately, tight lower bounds for DTW and SWM could greatly improve the efficiency by pruning large portions of non-candidate data at an early stage.

Comparisons of Distance Measures on Examples

Figure 1 demonstrates the effects of different distance measures with a typical piece of music, *Happy Birthday to You*, from top to bottom:

1. Since the query sequence is performed at a much faster tempo, direct application of DTW fails to produce an intuitive alignment;



Query by Humming. Figure 1. Motivating example.

2. Rescaling the shorter performance by a scaling factor of 1.54 seems to improve the alignment, but the higher pitched note produced on the third "*birth...*" of the candidate is forced to align with the lower note of the third "*happy...*" in the query;
3. Only the application of *both* uniform scaling and DTW produces the appropriate alignment.

Dynamic Time Warping (DTW)

Intuitively, *dynamic time warping* is a distance measure that allows time series to be *locally* stretched or shrunk before the base distance measure is applied. Given two sequences $C = C_1, C_2,...,C_n$ and $Q = Q_1, Q_2,...,Q_m$, the time warping distance DTW is defined recursively as follows:

$$\begin{aligned}
\mathrm{DTW}(\phi,\phi) &= 0\\
\mathrm{DTW}(C,\phi) = \mathrm{DTW}(\phi,Q) &= \infty\\
\mathrm{DTW}(C,Q) &= D_{base}(\mathrm{First}(C),\mathrm{First}(Q)) + \min\begin{cases}\mathrm{DTW}(C,\ \mathrm{Rest}(Q))\\ \mathrm{DTW}\,(\mathrm{Rest}(C),\ Q)\\ \mathrm{DTW}\,(\mathrm{Rest}(C),\mathrm{Rest}(Q))\end{cases}
\end{aligned}$$

where Φ is the empty sequence, $\mathrm{First}(C) = C_1$, $\mathrm{Rest}(C) = C_2, C_3,...,C_n$, and D_{base} denotes the distance between two entries. Several L_p measures were used as the D_{base} distance in previous literature, such as Manhattan Distance (L_1), squared Euclidean Distance (L_2) and maximum difference (L_∞). Typically *Squared Euclidean Distance* is used as the D_{base} measure. That is,

$$D_{base}(C_i, Q_j) = (C_i - Q_j)^2.$$

Thus in the following parts, without loss of generality, it is assumed that D_{base} is the squared Euclidean Distance and D is also used to denote it. However, the time complexity of DTW distance calculation is *O(mn)*, and intensive computations are employed for the corresponding dynamic programming. Thus, lower bounds on the distance are adopted to effectively prune the search space and support efficient search.

Constraints and Lower Bounds on Dynamic Time Warping

Keogh et al. [2] viewed a global constraint as a constraint on the warping path entry $w_k = (i, j)_k$ and gave a general form of global constraints in terms of inequalities on the indices to the elements in the warping matrix,

$$j - r \le i \le j + r$$

where r is a constant for the Sakoe–Chiba Band and r is a function of i for the Itakura Parallelogram. Incorporating the global constraint into the definition of dynamic time warping distance, DTW can be modified as follows.

Given two sequences $C = C_1,C_2,...,C_n$ and $Q = Q_1, Q_2,...,Q_m$, and the time warping constraint r, the constrained time warping distance cDTW is defined recursively as follows:

$$\begin{aligned}
Dist_r(C_i, Q_j) &= \begin{cases} D_{base}(C_i, Q_j) & if\,|\,i - j\,| \le r\\ \infty & otherwise\end{cases}\\
{}_C\mathrm{DTW}(\phi,\phi,r) &= 0\\
{}_C\mathrm{DTW}(C,\phi,r) = {}_C\mathrm{DTW}(\phi,Q,r) &= \infty\\
{}_C\mathrm{DTW}(C,Q,r) &= Dist_r(\mathrm{First}(C),\ \mathrm{First}(Q)) + \min\begin{cases}{}_C\mathrm{DTW}((C,\ \mathrm{Rest}(Q),r)\\ {}_C\mathrm{DTW}(\,(\mathrm{Rest}(C),\ Q,\ r)\\ {}_C\mathrm{DTW}((\mathrm{Rest}(C),\ \mathrm{Rest}(Q),r)\end{cases}
\end{aligned}$$

where Φ is the empty sequence, $\mathrm{First}(C) = C_1$, $\mathrm{Rest}(C) = C_2, C_3,...,C_n$. The *upper bounding sequence UW* and the *lower bounding sequence LW* of a sequence C are defined using the time warping constraint r as follows.

Let $UW = UW_1, UW_2,...,UW_n$ and $LW = LW_1, LW_2,...,LW_n$,

$$\begin{aligned}
UW_i &= \max(C_{i-r},...,C_{i+r}) \text{ and}\\
LW_i &= \max(C_{i-r},...,C_{i+r})
\end{aligned}$$

Considering the boundary cases, the above can be rewritten as

$$\begin{aligned}
UW_i &= \max(C_{\max(1,i-r)},...,C_{\min(i+r,n)}) \text{ and}\\
LW_i &= \min(C_{\min(1,i-r)},...,C_{\min(i+r,n)})
\end{aligned}$$

$E(C) = < UW, LW >$ is called the envelope sequences of Keogh et al. [2] propose the lower bound distance LB_Keogh based on envelope sequences. The time warping distance between two sequences Q and C is lower bounded by the squared Euclidean distance between Q and the envelope sequences of C. Equation (1) below formally defines the lower bounding distance.

$$\begin{aligned}
\mathrm{LB\_Keogh}(Q,\ C) &= D(Q,\ E(C))\\
&= \sum_{i=1}^{m}\begin{cases}(Q_i - UW_i)^2\ if & Q_i > UW_i\\ (Qi - LW_i)^2\ if & Q_i < LW_i\\ 0 & \text{otherwise}\end{cases} \qquad (1)
\end{aligned}$$

Zhu et al. [3] further improve on LB_Keogh. If a transformation T is a linear transform and lower-bounding, and $Env_r(C_i)$ is the envelope of C_i by global constraint r then

$$D(T(Q),\ T(Env_r(C_i))) \leq \mathrm{cDTW}(\mathrm{Q},\ \mathrm{C_i},\ \mathrm{r}) \quad (2)$$

Therefore transforms such as PAA, DWT, SVD and DFT on the envelope sequence of a candidate sequence could still lower bound DTW distance, since those transforms are both linear and lower bounding.

Uniform Scaling (US)

Given two sequences $Q = Q_1,\ldots,Q_m$ and $C = C_1,\ldots,C_n$ and a scaling factor bound $l, l \geq 1$. Let $C(q)$ be the prefix of C of length q, where $\lceil m/l \rceil \leq q \leq lm$ and $C(m,q)$ be a rescaled version of $C(q)$ of length m,

$$C(m,\ q)_i = C(q)_{\lceil i.q/m \rceil} \text{ where } 1 \leq \mathrm{i} \leq \mathrm{m}$$

$$\mathrm{US}(C, Q, l) = \min_{q=\lceil m/l \rceil}^{min(lm,n)} D(C(m,q), Q)$$

where D(X, Y) denotes the Euclidean distance between two sequences X and Y.

Lower Bounding Uniform Scaling

The two sequences $UC = UC_1,\ldots,UC_m$ and $LC = LC_1,\ldots,LC_m$, such that

$$UC_i = \max(C_{\lceil i/l \rceil},\ldots,C_{\lceil il \rceil})$$
$$LC_i = \min(C_{\lceil i/l \rceil}),\ldots,C_{\lceil il \rceil})$$

bound the points of the time series C that can be matched with Q. The lower bounding function, which lower bounds the distance between Q and C for any scaling ρ, $1 \leq \rho \leq l$, can now be defined as:

$$LB_s(Q,C) = \sum_{i=1}^{m} \begin{cases} (Q_i - UC_i)^2 & \text{if } Q_i > UC_i \\ (Q_i - LC_i)^2 & \text{if } Q_i < UC_i \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Scaling and Time Warping (SWM)

Having reviewed time warping, uniform scaling, and lower bounding, this part introduces *scaling and time warping* (SWM) distance. Given two sequences $Q = Q_1,\ldots,Q_m$ and $C = C_1,\ldots,C_n$, a bound on the scaling factor $l, l \geq 1$ and the Sakoe–Chiba Band time warping constraint r which applies to sequence length m. Let $C(q)$ be the prefix of C of length q, where $\lceil m/l \rceil \leq q \leq \min(lm, n)$ and $C(m, q)$ be a rescaled version of $C(q)$ of length m,

$$\mathrm{C}(m,q)i = C(q)_{\lceil i.q/m \rceil} \text{ where } 1 \leq i \leq m$$

$$\mathrm{SWM(C,Q,l,r)} = \min_{\mathrm{q}=\lceil \mathrm{m/l} \rceil}^{\mathrm{min(lm,n)}} \mathrm{cDTW(C(m,q),Q,r)}$$

If time warping is applied on top of scaling, i.e., the sequence is first scaled, and then measure the time warping distance of the scaled sequence with the query. Typically, time warping with Sakoe–Chiba Band constrains the warping path by a fraction of the data length, which is translated into a constant r. Hence, if the fraction is 10%, then $r = 0.1|C|$. If the length of C is changed according to the scaling fraction ρ, that is, C is changed to ρC, then the Sakoe–Chiba Band time warping constraint is $r = 0.1|\rho C|$. Hence, $r = r'\rho$, where r' is the Sakoe–Chiba Band time warping constraint on the unscaled sequence, and ρ is the scaling factor.

Lower Bounding SWM

The lower envelope L_i and upper envelope U_i on C can be deduced as follows: recall that the upper and lower bounds for uniform scaling between $1/l$ and l is given by the following:

$$UC_i = \max(C_{\lceil i/l \rceil},\ldots,C_{\lceil il \rceil})$$
$$LC_i = \min(C_{\lceil i/l \rceil},\ldots,C_{\lceil il \rceil})$$

and the upper and lower bounds for a Sakoe –Chiba Band time warping constraint factor of r for a point C_i is given by:

$$UW_i = \max(C_{\max(1,i-r)},\ldots,C_{\min(i+r,n)})$$
$$LW_i = \min(C_{\max(1,i-r)},\ldots,C_{\min(i+r,n)})$$

Therefore, when time warping is applied on top of scaling the upper and lower bounds will be:

$$\begin{aligned} U_i &= \max(\mathrm{UW}_{\lceil i/l \rceil},\ldots,UW_{\lceil il \rceil}) \\ &= \max(C_{\max(1,\lceil i/l \rceil - r')},\ldots,C_{\min(\lceil i/l \rceil + r',n)},\ldots, \\ &\qquad C_{\max(1,\lceil il \rceil - r')},\ldots,C_{\min(\lceil il \rceil + r',n)}) \\ &= \max(C_{\max(1,\lceil i/l \rceil - r')},\ldots,C_{\min(\lceil il \rceil + r',n)}) \end{aligned} \quad (4)$$

$$\begin{aligned} L_i &= \min(\mathrm{LW}_{\lceil i/l \rceil},\ldots,LM_{\lceil il \rceil}) \\ &= \min(C_{\max(1,\lceil i/l \rceil - r')},\ldots,C_{\min(\lceil i/l \rceil + r',n)},\ldots, \\ &\qquad C_{\max(1,\lceil il \rceil - r')},\ldots,C_{\min(\lceil il \rceil + r',n)}) \\ &= \min(C_{\max(1,\lceil i/l \rceil - r')},\ldots,C_{\min(\lceil il \rceil + r',n)}) \end{aligned} \quad (5)$$

In [4], the lower bound function which lower bounds the distance between Q and C for any scaling in the range of $\{1/l, l\}$ and time warping with the Sakoe–Chiba Band constraint factor of r' on C is given by:

$$LB(Q,C) = \sum_{i-1}^{m} \begin{cases} (Q_i - U_i)^2 & \text{if } Q_i > U_i \\ (Q_i - L_i)^2 & \text{if } Q_i < L_i \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Efficient Pruning Algorithm by Lower Bounds

Algorithm 1 gives the pseudocode for the search algorithm, which utilizes the computational efficient lower bounds on computational intensive distance measures to prune candidate sequences at an early stage. "real_distance" could be DTW, US, or SWM distance, while "lower_bound_distance" denotes the corresponding lower bound.

```
Algorithm 1: Lower_Bounding_Sequential_Scan(Q)
best_so_far = infinity;
for each sequence with index i in database do
   LB_dist = lower_bound_distance($C_i$, $Q$);
   if LB_dist < best_so_far then
      true_dist = real_distance($C_i$, $Q$);
      if true dist < best_so_far then
         best_so_far = true_dist;
         index_of_best_match = $i$;
      end
   end
end
```

Key Applications

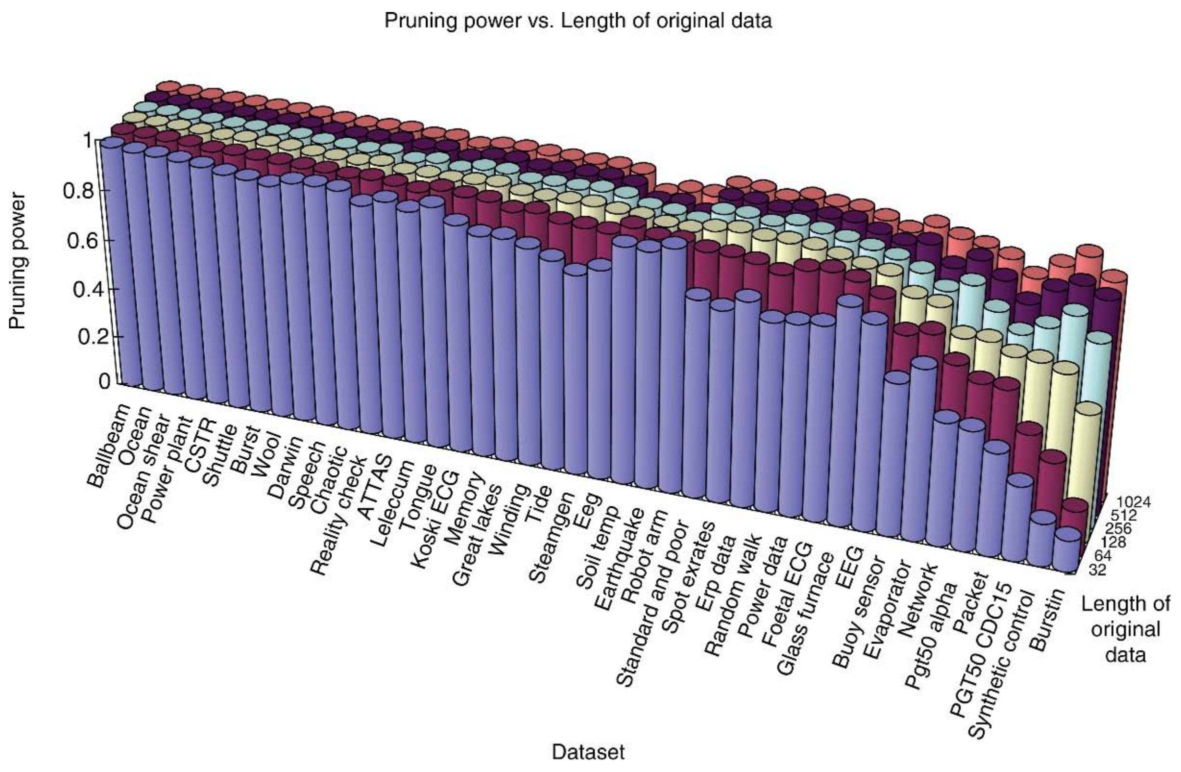
Query by humming is essential for audio information retrieval in terms of both effectiveness and efficiency.

Future Directions

In [3], a unified framework is proposed to explain the existing lower-bound functions for dynamic time warping distance. A new lower-bound function that is shown by experiments to be superior is also proposed. For future studies, this function can be investigated for the effectiveness in query by humming.

Experimental Results

This section describes the experiments carried out to verify the effectiveness of the proposed lower bounding



Query by Humming. Figure 2. Pruning power vs. length of original data.

distance for the most effective distance measure: SWM. The *Pruning Power P* is defined in [2] as follows:

$$p = \frac{\text{Number of objects that do not require full SWM}}{\text{Number of objects in database}}$$

The pruning power is an objective measure because it is free of implementation bias and choice of the underlying spatial index. This measure has become a common metric for evaluating the efficiency of lower bounding distances, therefore, it was adopted in evaluating the proposed lower bounding distance.

Figure 2 shows how the pruning power of the lower bounding measure varies as the length of data changes for different datasets. More than 78% (32 out of 41) of the datasets achieved a pruning power above 90%.

Data Sets

http://www.cs.ucr.edu/∼eamonn/VLDB2005/

Cross-references

▶ Multimedia Information Retrieval
▶ Spatial Network Databases

Recommended Reading

1. Ghias A., Logan J., Chamberlin D., and Smith B.C. Query by humming: musical information retrieval in an audio database. In Proc. 3rd ACM Int. Conf. on Multimedia, 1995, pp. 231–236.
2. Keogh E.J. Exact indexing of dynamic time warping. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 406–417.
3. Zhou M. and Hon Wong M. Boundary-based lower-bound functions for dynamic time warping and their indexing. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 1307–1311.
4. Zhu Y. and Shasha D. Warping indexes with envelope transforms for query by humming. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 181–192.
5. Wai-Chee Fu A., Keogh E.J., Yung Hang Lau L., and Chotirat (Ann) Ratanamahatana. Scaling and time warping in time series querying. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 649–660.

Query Compilation

▶ Query Optimization
▶ Query Optimization (in Relational Databases)

Query Compilation and Execution

▶ Query Processing (In Relational Databases)

Query Containment

Rada Chirkova
North Carolina State University, Raleigh, NC, USA

Definition

One query is contained in another if, independent of the values of the "stored data" (that is, database), the set of answers to the first query on the database is a subset of the set of answers to the second query on the same database. A formal definition of containment is as follows: denote with $Q(D)$ the result of computing query Q over database D. A query Q_1 is said to be contained in a query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all databases D, the set of tuples $Q_1(D)$ is a subset of the set of tuples $Q_2(D)$ – that is, $Q_1(D) \subseteq Q_2(D)$. This definition of containment, as well as the related definition of query equivalence, can be used to specify query containment and equivalence on databases conforming to both relational and nonrelational data models, including XML and object-oriented databases.

Historical Background

Testing for query containment on finite databases is, in general, co-recursively enumerable: The procedure is going through all possible databases and simultaneously checking for noncontainment via bottom-up evaluation. See [6] for the details and for a discussion of the relationship between regular and unrestricted (that is, on not necessarily finite databases) containment and equivalence of relational expressions.

Chandra and Merlin [3] have shown that the problems of containment, equivalence, and minimization of conjunctive queries are NP complete. (Conjunctive queries are a subset of Datalog that is equivalent in expressive power to SQL select-project-join queries with only equality comparisons permitted.) It is also shown [3] that there is a simple test for containment, and thus for equivalence. While the question of whether one conjunctive query is contained in another is NP complete, all the complexity is caused by "repeated predicates," that is, predicates appearing three or more times in the body. In the very common case that no predicate appears more than twice in any query, containment can be tested in linear time [11]. Moreover, conjunctive queries tend to be short, so in practice containment testing is not likely to be too inefficient.

Klug [8] has shown that containment for conjunctive queries with arithmetic – that is, inequality or disequality – comparisons (CQAC) is in $\Pi_2{}^P$, while being in NP for some proper subclasses. In addition to reporting some new results on the complexity of CQAC query containment, Afrati and colleagues [2] provide a survey, comprehensive as of 2006, on the complexity of query containment, for query classes including conjunctive queries, CQAC and its subclasses, as well as recursive and nonrecursive Datalog. More original work and further references on query containment in relational and nonrelational databases can be found in [1,5,7,9,10,12–15].

Foundations

This overview of key ideas concerning conjunctive queries, Datalog programs, and their containment is based on [9] as well as on [15], which includes the details and references to the original sources of the results discussed here. The first item on the agenda is a review of containment of conjunctive queries. A conjunctive query is a single Datalog rule with subgoals that are assumed to have predicates referring to stored relations. (The standard Datalog notation is reviewed in, e.g., Sect. 6.1 of [4].) A conjunctive query is applied to the stored relations in a given database by considering all possible substitutions of values for the variables in the body. If a substitution makes all the subgoals true, then the same substitution, applied to the head of the rule, is an element of the set of answers to the head's predicate on the given database.

For example, rule

$$p(X,Z)\ :-\ a(X,Y),\ a(Y,Z)$$

talks about predicate a, for stored relation that contains information about arcs in a directed graph: $a(X, Y)$ means that there is an arc from node X to node Y in the graph. The rule also talks about predicate p whose relation is constructed by the rule. The rule says "$p(X, Z)$ is true if there is an arc from node X to some node Y and also an arc from Y to Z." That is, conjunctive query p represents paths of length 2, in the sense that $p(X, Z)$ will be inferred exactly when there is a path of length 2 from X to Z in the graph represented by stored relation a.

It was proved in [3] that $Q_1 \sqsubseteq Q_2$ if and only if there is a homomorphism $h : Q_2{}^D \rightarrow Q_1{}^D$, where Q^D is the canonical database associated with conjunctive query Q. The canonical database Q^D for query Q is defined as the result of "freezing" the body of Q, which turns each subgoal of Q into a fact in the database. That is, the "freezing" procedure replaces each variable in the body of Q by a distinct constant, and the resulting subgoals are the only tuples in the canonical database Q^D.

Consider now an example of checking conjunctive-query containment using canonical databases. Rule

$$r(W,W)\ :-\ a(W,U),\ a(U,W)$$

defines conjunctive query r, whose answer represents circular paths of length 2 in a graph specified by arcs stored in relation a. As each circular path of length 2 is also an (arbitrary) path of length 2, the containment $r \sqsubseteq p$ is expected to hold. Indeed, the canonical database r^D, for the query defining predicate r, is a set of tuples $\{a(w, u), a(u, w)\}$, while the canonical database p^D, for the query defining predicate p, is a set of tuples $\{a(x, y), a(y, z)\}$. There exists a homomorphism from p^D to r^D that maps x into w, y into u, and z into w (or, alternatively, maps $a(x, y)$ into $a(w, u)$ and $a(y, z)$ into $a(u, w)$). Thus, by [3], the set of tuples in the answer to r on every database D is a subset of the set of tuples in the answer to p on D.

As no homomorphism exists from r^D to p^D, the conclusion is that p is not contained in r, which is to be expected from an intuitive interpretation of the two queries. However, if conjunctive query t defined by the rule

$$t(L,N)\ :-\ a(L,M),\ a(M,N),\ a(M,S)$$

is also considered, it is possible to ascertain both $p \sqsubseteq t$ and $t \sqsubseteq p$, by constructing a homomorphism from p^D to t^D (for the query defining predicate t) and another homomorphism from t^D to p^D. Thus, conjunctive queries p and t are *equivalent*. Indeed, each of p and t represents arbitrary paths of length 2 in a graph. However, p is more efficient to evaluate than t, because computing the set of answers to p requires only one join on the stored relation a, while evaluating t requires two joins. As described in [3], one can *minimize* conjunctive queries for more efficient evaluation; in fact, query p in this example can be obtained as a result of minimizing query t.

Another test for containment of conjunctive query Q_1 in conjunctive query Q_2 consists in computing the set of answers to Q_2 on the canonical database $Q_1{}^D$ for Q_1. The test succeeds if the frozen head of Q_1 is an element of $Q_2(Q_1^D)$; otherwise the database Q_1^D is a counterexample to the containment.

An important extension of the theory of containment of conjunctive queries is the inclusion of arithmetic comparisons as subgoals, with so-called built-in, or interpreted, predicates (e.g., subgoal $X \leq Y$ with built-in predicate $\leq$). When testing two conjunctive queries with arithmetic comparisons (CQACs) for containment $Q_1 \sqsubseteq Q_2$ using canonical databases, one must consider the set of values in the database as belonging to a totally ordered set, e.g., the integers or reals.

The containment test using canonical databases is conducted as follows. Each basic canonical database is constructed from only those subgoals of Q_1 that have uninterpreted predicates. Each basic canonical database is the canonical database (Q^D for query Q) defined above, together with a partition of the variables of the query into a list of blocks, such that each block is associated with a distinct integer value for all its variables, in increasing order of the integer values on the list of the blocks. The containment test succeeds if the frozen head of Q_1 is an element of $Q_2(Q_1^{D_i})$ on *all* basic canonical databases $Q_1^{D_i}$ for Q_1; otherwise any $Q_1^{D_i}$ on which the test fails is a counterexample to containment. Another more general containment test for conjunctive queries with interpreted (not necessarily arithmetic-comparison) predicates uses homomorphisms on the uninterpreted predicates and logical implication on the built-in predicates; see [15] for the details.

The problem of testing CQACs for containment is complete for Π_2^P, at least in the case of a dense domain such as real numbers. (In fact, the problem is Π_2^P complete even for conjunctive queries with disequalities $\neq$ as the only comparison predicate.) A containment test for conjunctive queries with negation is outlined in [15]. The test in this case is also complete for Π_2^P, as it also involves exploring an exponential number of canonical databases. The query-containment problem is also $\mathit{\Pi}_2^P$ complete for unions of conjunctive queries [1].

Containment questions involving Datalog programs are often harder than for conjunctive queries. It is known that containment of Datalog programs is undecidable, while containment of a Datalog program in a conjunctive query is doubly exponential. However, the important case for purposes of information integration is the containment of a conjunctive query in a Datalog program, and this question turns out to be no more complex than containment of conjunctive queries. To test whether a conjunctive query Q is contained in a Datalog program P, one would "freeze" the body of Q to make a canonical database D. Then a check is done to see if $P(D)$ contains the frozen head of Q. The only significant difference between containment in a conjunctive query and containment in a Datalog program is that in the latter case one must keep applying the rules until either the head of Q is derived or no more facts can be inferred in evaluating $P(D)$.

Key Applications

Query containment was recognized fairly early as a fundamental problem in database query evaluation and optimization. The reason is, for conjunctive queries – a broad class of frequently used queries, whose expressive power is equivalent to that of select-project-join queries in relational algebra – query containment can be used as a tool in query optimization, since the problem of conjunctive-query equivalence is equivalent to the problem of conjunctive-query containment. Specifically, to find a more efficient *and* answer-preserving formulation of a given conjunctive query, it is enough to "try all ways" of arriving at a "shorter" query formulation, by removing a query subgoal, in a process called query minimization [3]. In this process, a subgoal-removal step succeeds only if a containment test ensures equivalence (via containment) of the "original" and "shorter" query formulations.

Note that the problems of query containment and query equivalence are equivalent for conjunctive queries under the common setting of *set semantics for query evaluation,* where both stored relations and query answers are interpreted as sets of tuples. Interestingly, the relationship between the problems of containment and equivalence is very different under *bag semantics for query evaluation,* where both stored relations and query answers are allowed to have duplicates. See Jayram and colleagues [5] for a discussion and references on containment and equivalence under bag semantics, for conjunctive queries as well as for more expressive classes of queries, including CQACs and queries with grouping and aggregation. Jayram and colleagues [5] also present original undecidability results for containment of conjunctive queries with inequalities under bag and bag-set semantics for query evaluation.

In recent years, there has been renewed interest in the study of query containment, because of its close relationship to the problem of answering queries using views [4]. Intuitively, the problem of answering queries using views is as follows: Given a query on a database schema and a set of views (i.e., named queries) over the same schema, can the query be answered (efficiently) using only the views? Alternatively, what is the maximum set of tuples in the answer to the query that can be obtained from the views? As another alternative, in case it is possible to access both the views and the database relations, what is the cheapest query-execution plan for answering the query?

The problem of answering queries using views has emerged as a central problem in integrating information from heterogeneous sources, an area that has been the focus of concentrated research efforts for a number of years [4,15]. An information-integration system can be described logically by views that specify what queries the various information sources can answer. These views might be conjunctive queries or Datalog programs, for example. The "database" of predicates over which these views are defined is not a concrete database but rather a collection of "global" predicates whose actual values are determined by the sources, via the views. Information-integration systems provide a uniform query interface to a multitude of autonomous data sources, which may reside within an enterprise or on the World-Wide Web. Data-integration systems free the user from having to locate sources relevant to a query, interact with each one in isolation, and manually combine data from multiple sources.

Given a user query Q, typically a conjunctive query, on the global predicates, an information-integration system determines whether it is possible to answer Q by using the various views in some combination. In addressing the problem of answering queries using views in the information-integration setting, query containment appears to be more fundamental than query equivalence. In fact, answering a query using only the answers to the views is considered "good enough" even in cases where equivalence does not hold (or cannot be demonstrated), provided that the view-based query rewriting can be shown to be contained in the query and is a maximal (i.e., returning the maximal set of answers) rewriting of the query using the available views and a given rewriting language. (For the details and references on maximally contained rewritings see, e.g., [2].)

Besides its applications in information integration, the problem of answering queries using views is of special significance in other data-management applications. (Please see [4] for the details and references.) For instance, in query optimization finding a rewriting of a query using a set of materialized views (i.e., the answers to the queries defining the views) can yield a more efficient query-execution plan, because part of the computation necessary to answer the query may have already been done while computing the materialized views. Such savings are especially significant in decision-support applications when the views and queries contain grouping and aggregation.

In the context of database design, view definitions provide a mechanism for supporting the independence of the logical and physical views of data. This independence enables the developers to modify the storage schema of the data (i.e., the physical view) without changing its logical schema, and to model more complex types of indices. Provided the storage schema is described as a set of views over the logical schema, the problem of computing a query-execution plan involves figuring out how to use the view answers (i.e., the physical storage) to answer the query posed on the logical schema.

In the area of data-warehouse design the desideratum is to choose a set of views (and indexes on the views) to materialize in the warehouse. Similarly, in web-site design, the performance of a web site can be significantly improved by choosing a set of views to materialize. In both problems, the first step in determining the utility of a choice of views is to ensure that the views are sufficient for answering the queries expected to be posed over the data warehouse or the web site. The problem, again, translates into the view-rewriting problem.

The problem of query containment is also of special significance in artificial intelligence, where conjunctive queries, or similar formalisms such as description logic, are used in a number of applications. The design theory for such logics is reducible to containment and equivalence of conjunctive queries. Original results and a detailed discussion concerning an intimate connection between conjunctive-query containment in database theory and constraint satisfaction in artificial intelligence can be found in [9].

Cross-references

- ▶ Conjunctive Query
- ▶ Query Optimization
- ▶ Query Optimization (in Relational Databases)
- ▶ Query Rewriting Using Views
- ▶ SQL

Recommended Reading

1. Abiteboul S., Hull R., and Vianu V. Foundations of Databases. Addison-Wesley, 1995.
2. Afrati F.N., Li C., and Mitra P. Rewriting queries using views in the presence of arithmetic comparisons. Theor. Comput. Sci., 368(1–2):88–123, 2006.
3. Chandra A.K. and Merlin P.M. Optimal implementation of conjunctive queries in relational data bases. In Proc. 9th Annual ACM Symp. on Theory of Computing, 1977, pp. 77–90.
4. Halevy A.Y. Answering queries using views: A survey. VLDB J., 10(4):270–294, 2001.
5. Jayram T.S., Kolaitis P.G., and Vee E. The containment problem for REAL conjunctive queries with inequalities. In Proc. 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2006, pp. 80–89.
6. Kanellakis P.C. Elements of Relational Database Theory. In Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). Elsevier and the MIT Press, 1990, pp. 1073–1156.
7. Kimelfeld B. and Sagiv Y. Revisiting Redundancy and Minimization in an XPath Fragment. In Advances in Database Technology, Proc. 11th Int. Conf. on Extending Database Technology, 2008, pp. 61–72.
8. Klug A.C. On conjunctive queries containing inequalities. J. ACM, 35(1):146–160, 1988.
9. Kolaitis P.G. and Vardi M.Y. Conjunctive-Query Containment and Constraint Satisfaction. J. Comput. Syst. Sci., 61 (2):302–332, 2000.
10. Miklau G. and Suciu D. Containment and equivalence for a fragment of XPath. J. ACM, 51(1):2–45, 2004.
11. Saraiya Y. Subtree elimination algorithms in deductive databases. Ph.D. thesis, Stanford University, 1991.
12. Ullman J.D. CS345 lecture notes. http://infolab.stanford.edu/~ullman/cs345-notes.html.
13. Ullman J.D. Principles of Database and Knowledge-Base Systems, Volume II. Computer Science press, 1989.
14. Ullman J.D. The database approach to knowledge representation. In Proc. 13th National Conf. on Artificial Intelligence and 8th Innovative Applications of AI Conf., Volume 2, 1996, pp. 1346–1348.
15. Ullman J.D. Information integration using logical views. Theor. Comput. Sci., 239(2):189–210, 2000.

Query Engine

► Query Processor

Query Evaluation

► Evaluation of Relational Operators
► Query Processing
► Query Processing and Optimization in Object Relational Databases

Query Evaluation Plan

► Query Plan

Query Evaluation Techniques for Multidimensional Data

Amarnath Gupta
University of California-San Diego, La Jolla, CA, USA

Synonyms

Spatial data

Definition

There are two senses of the term "multidimensional data." The first relates to the data warehousing and online analytical processing (OLAP). The second sense of the term, used mostly in the context of scientific data, refers to variants of array-representable data where the dimensionality refers to the dimensions of the array. Query processing for this class of data uses array algebras [3] and array-specific storage [1] indexing techniques [4].

Key Points

In many scientific applications, data can be represented as multidimensional arrays. For example, the current flow in oceans is a time-varying vector field and can be roughly viewed as 4-dimensional data. In many applications the array may not be uniform, but it can be nested, and even irregular. Query evaluation on this kind of data is greatly dependent on applications. Some applications need to perform operations like value-based clustering on the data, while other applications need fast computation of aggregates such as temporal trends in regions that are selected by user queries. It has been established that storing the multidimensional data as relational tables and developing special routines to handle such relational data is feasible but not optimal, especially when the data is large. ESRI, the GIS provider, has assembled a number of operations collectively called the MapAlgebra for the case where the arrays are two dimensional and the applications are spatial. Recently [2] developed a generic algebra called the gridfield algebra for manipulating arbitrary gridded datasets (i.e., arrays), and present algebraic optimization techniques in these applications. This system is implemented in a system called CORIE (http://www.ccalmr.ogi.edu/CORIE/) for an environmental observation and forecasting system.

Cross-references

▶ Multidimensional Modeling
▶ On-Line Analytical Processing
▶ Raster Data Management and Multi-Dimensional Arrays

Recommended Reading

1. Furtado P. and Baumann P. Storage of multidimensional arrays based on arbitrary tiling. In Proc. 15th Int. Conf. on Data Engineering, 1999, pp. 480–489.
2. Howe B. and Maier D. Algebraic manipulation of scientific datasets. VLDB J., 14(4):397–416, 2005.
3. Marathe A.P. and Salem K. Query processing techniques for arrays. VLDB J. 11(1):68–91, 2002.
4. Sinha R.R. and Winslett M. Multi-resolution bitmap indexes for scientific data. ACM Trans. Database Syst., 32(3):16, 2007.

Query Execution Engine

▶ Query Processor

Query Execution in Star/ Snowflake Schemas

▶ Query Processing in Data Warehouses

Query Execution Plan

▶ Query Plan

Query Expansion

▶ Web Search Query Rewriting

Query Expansion for Information Retrieval

Olga Vechtomova
University of Waterloo, Waterloo, ON, Canada

Synonyms

QE, Query enhancement; Term expansion

Definition

Query expansion (QE) is a process in Information Retrieval which consists of selecting and adding terms to the user's query with the goal of minimizing query-document mismatch and thereby improving retrieval performance.

Historical Background

The work on query expansion following relevance feedback dates back to 1965, when Rocchio [9] formalized relevance feedback in the vector-space model. Early work on using collection-based term co-occurrence statistics to select query expansion terms was done by Spärck Jones [10] and van Rijsbergen [12].

Foundations

The central task of information retrieval (IR) is to find documents that satisfy the user's information need. This is usually taken to mean finding documents or some parts of them, such as passages, which contain information that would help resolving the user's information need. Therefore, at least in a more traditional sense, IR does not involve providing the user directly with the information needed. The user usually expresses his/her information need in free-text using natural language words and phrases (terms). Sometimes, prior to retrieving the documents, the user's free-text query is translated into controlled vocabulary which is a subset of the words that comprise a natural language. A document is similarly indexed either by the natural language terms that constitute its content, or by a set of controlled index terms that map its contents to the concepts in a given domain. Both directly matching free-text query terms to free-text index terms and translating natural language words to controlled vocabularies are inherently imprecise. In the first case, the main problem is that the user and the author of a document may express the same idea by means of different terms. In the second case, the shades of meanings that natural language terms carry may be lost in the translation process. In addition to these problems, the user's query may be incomplete or inaccurate, i.e., the user may not specify his/her information need exactly or express it accurately.

The goal of query expansion is to enrich the user's query by finding additional search terms, either automatically, or semiautomatically that represent the user's information need more accurately and completely, thus avoiding, at least to an extent, the aforementioned problems, and increasing the chances of matching the user's query to the representations of relevant ideas in documents. Query expansion techniques may be categorized by the following criteria:

- Source of query expansion terms;
- Techniques used for weighting query expansion terms;
- Role and involvement of the user in the query expansion process.

Query expansion can be performed automatically or interactively. In automatic query expansion (AQE), the system selects and adds terms to the user's query, whereas in interactive query expansion (IQE), the system selects candidate terms for query expansion, shows them to the user, and asks the user to select (or deselect) terms that they want to include into (or exclude from) the query.

There are three main sources of QE terms: (i) hand-built knowledge resources such as dictionaries, thesauri, and ontologies; (ii) the documents used in the retrieval process; (iii) external text collections and resources (e.g., the WWW, Wikipedia).

Hand-built knowledge resources have three main limitations: they are usually domain-specific, have to be kept up-to-date, and typically do not contain proper nouns. Experiments with QE using knowledge resources did not show consistent performance improvements. For example, in [13] QE with words manually selected from WordNet, a large domain-independent lexical resource with lexical-semantic relations between words, did not improve well-formulated queries, but significantly improved performance of poorly-constructed queries.

The most common source of QE terms is the text collection used in the retrieval process or its subset (e.g., retrieved documents). These QE techniques showed overall better performance than techniques using hand-built knowledge resources. They can be subdivided into the following categories:

– QE following relevance feedback. QE terms are extracted from the documents retrieved in response to the user's query and judged relevant by the user.
– QE following blind (pseudo-relevance) feedback. QE terms are extracted from the top-ranked documents retrieved in response to the user's query.
– QE using automatically built association thesauri and collection-wide word co-occurrences.

Query Expansion Following Relevance and Pseudo-Relevance Feedback

Relevance feedback (RF) is a process by which the system, having retrieved some documents in response to the user's query, asks the user to assess their relevance to his/her information need. Documents are typically shown to the user in some surrogate form, for example, as document titles, abstracts, snippets of text, query-biased, or general summaries, keywords and key-phrases. The user may also have an option to see the whole document before making the relevance judgement. After the user has selected some documents as relevant, query expansion terms are extracted from them, weighted and the top-weighted terms are either added to the query automatically, or shown to the user for further selection.

Query expansion following relevance feedback has consistently yielded substantial gains in performance in experimental settings. Many term selection methods have been proposed for query expansion following RF. The general idea behind all such methods is to select terms that will be useful in retrieving previously unseen relevant documents. Below is a brief description of a query expansion method [11] which showed consistently high performance results on Text REtrieval Conference (TREC) test collections. The first step is to retrieve documents in response to the user's initial query, which is done by calculating document matching score (Eq. 1) using the Robertson/Spärck-Jones probabilistic model.

$$MS = \sum_{i \in Q} \frac{(k_1 + 1) \times tf_i}{k_1 \times NF + tf_i} \times w_i \qquad (1)$$

Where i is a term in the query Q, tf_i is the frequency of i in the document, k_1 is term frequency normalization factor, NF is document length normalization factor and is calculated as $NF = (1\text{-}b) + b \times DL/AVDL$, where DL is document length, $AVDL$ is average document length, b is a tuning constant, w_i is term collection weight, calculated as $w_i = log(N/n_i)$, where N is the number of documents in the collection, n_i is the number of documents containing i.

After the user looks through the top-retrieved documents, and judges some of them as relevant, the system extracts all terms from these documents, ranks them according to the Offer Weight (OW) in Eq. 2, and either adds a fixed number of terms to the query (automatic query expansion), or asks the user to perform the term selection.

$$OW = r \times RW \tag{2}$$

Where r is the number of relevant documents containing the candidate query expansion term and RW is Relevance Weight calculated as shown in Eq. 3:

$$RW = \log\left(\frac{(r+0.5)(N-n-R+r+0.5)}{(R-r+0.5)(n-r+0.5)}\right) \tag{3}$$

Where R is the number of documents judged relevant; r is the same as above; N is the number of documents in the collection; n is the number of documents containing the term.

The subsequent document retrieval with the expanded query is performed using Eq. 1 for document ranking with RW used instead of w.

A related approach to RF is pseudo-relevance or blind feedback (BF), which uses a number of top-ranked documents in the initially retrieved set for query expansion without asking the user to assess their relevance. The query expansion method described above can be used in BF with R being the number of top documents *assumed* to be relevant. Many other QE methods following blind feedback have been proposed. Two of the methods that showed good performance on large test collections are briefly introduced below.

Local Context Analysis (LCA) [14] technique consists of extracting terms (nouns and noun phrases) from n top ranked passages retrieved in response to the user's query. The extracted terms are ranked by their similarity to the entire user's query, and top m terms are added to the query. Carpineto et al. [3] proposed a term ranking method for QE based on Kullback-Leibler divergence (KLD) measure. The method ranks candidate QE terms based on the difference between their distribution in pseudo-relevant documents and in the entire collection.

In general, blind feedback has been demonstrated to be less robust in performance than relevance feedback. The QE performance following BF depends greatly on the performance of the user's initial query: if many of the top-ranked documents retrieved in response to the initial query are relevant, then it is likely that QE terms from these documents will be useful in retrieving other relevant documents. However, if the initial query is poorly formulated or ambiguous, and many top-ranked documents are nonrelevant, then QE terms extracted from such documents may deteriorate performance. Billerbeck and Zobel [2] report that blind feedback in their experiments improves performance of less than a third of queries. They also conclude that the best values for such BF parameters as the number of documents and terms used for QE vary widely across topics.

Query Expansion Using Association Thesauri and Term Co-Occurrence Measures

Unlike QE following relevance or pseudo-relevance feedback, where terms are selected from documents at search time, QE techniques in this category rely on lexical resources automatically constructed prior to the search process. Typically, statistical measures of term similarity are applied to identify terms in a large document collection that co-occur in the same contexts, and which, therefore, are likely to be conceptually related.

For example, Qiu and Frei [7] developed a query expansion method where query expansion terms are selected from an automatically constructed co-occurrence based term-term similarity thesaurus on the basis of the degree of their similarity to all terms in the query. Jing and Croft [4] developed a technique for automatic construction of a co-occurrence thesaurus. Each indexing unit, defined through a set of phrase rules, is recorded in the thesaurus with its most strongly associated terms. An evaluation of different similarity measures (Dice, Jaccard, Cosine, Average Conditional Probability, and Normalized Mutual Information) for selecting query expansion terms from a document collection is reported in [5]. The Dice, Jaccard, and Cosine led to better QE results than Average Conditional Probability and Normalized Mutual Information.

Interactive Query Expansion

In interactive query expansion the task of selecting and adding terms to the user's query is split between the user and the system in such a way that the system selects the candidate query expansion terms, but it is the user who makes the final decision which terms to include into the query. Similarly to automatic query expansion, the most common source of terms used in IQE is a set of documents resulting from either relevance, or pseudo-relevance feedback. Terms are extracted by the system, ranked, and the top-ranked terms are shown to the user for selection. The process of IQE is iterative, and may be triggered either by the system, or the user.

Several studies comparing the effectiveness of AQE and IQE have been conducted. Intuitively, since the user is the one who decides which document is relevant to his/her information need, the user should be able to make better decisions than the system with respect to which terms to add to the query. However, experiments do not offer conclusive results that IQE is more effective than AQE. For instance, Beaulieu [1] showed that AQE is more effective than IQE in operational settings. On the other hand, Koenemann and Belkin [6] reported higher subjective user satisfaction with an IQE system, as well as better performance. This suggests that the effectiveness of IQE is highly variable, depending on the specific interactive system, the users, and the task. Ruthven [8] did a simulation study of a potential benefit of IQE. He concludes that while IQE has potential to achieve higher performance than AQE, this potential may not be easy to realize, because it is difficult for the users to make decisions about which terms are better in differentiating between relevant and nonrelevant documents.

Key Applications

Query expansion is used in some web search engines and enterprise search systems. Although QE following relevance feedback has been experimentally shown as one of the most successful IR techniques, its use in Web search has been limited.

Cross-references

► BM25
► Information Retrieval
► Information Retrieval Models
► Probabilistic Retrieval Models and Binary Independence Retrieval (BIR) Model.
► Query Expansion Models
► Relevance Feedback for Text Retrieval

Recommended Reading

1. Beaulieu, M. Experiments with interfaces to support query expansion. J. Doc. 53(1):8–19, 1997.
2. Billerbeck B. and Zobel J. Questioning query expansion: an examination of behaviour and parameters. In Proc. 15th Australasian Database Conf., 2004, pp. 69–76.
3. Carpineto C., de Mori R., Romano G., and Bigi B. An information-theoretic approach to automatic query expansion. ACM Trans. Information Syst., 19(1):1–27, 2001.
4. Jing Y. and Croft B. An association thesaurus for information retrieval. In Proc. 4th Int. Conf. Computer-Assist IR. "Recherche d'Information Assistée par Ordinateur", pp. 146–160.
5. Kim M-C. and Choi K-S. A comparison of collocation-based similarity measures in query expansion. Inf. Proc. & Man., 35, 19–30, 1999.
6. Koenemann J. and Belkin N.J. A case for interaction: a study of interactive information retrieval behavior and effectiveness. In Proc. SIGCHI Conf. on Human Factors in Computing Systems, 1996, pp. 205–212.
7. Qiu Y. and Frei H.P. Concept based query expansion. In Proc. 16th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1993, pp. 160–169.
8. Ruthven I. Re-examining the potential effectiveness of interactive query expansion. In Proc. 26th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 2003, pp. 213–220.
9. Salton G. The SMART retrieval system (Chapter 14). Prentice-Hall, Englewood Cliffs NJ. (Reprinted from Rocchio J.J. (1965). Relevance feedback in information retrieval. In Scientific Report ISR-9, Harvard University), 1971.
10. Spärck Jones K. Automatic keyword classification for information retrieval. Butterworths, London, 1971.
11. Spärck Jones K., Walker S., and Robertson S.E. A probabilistic model of information retrieval: development and comparative experiments. Inf. Proc. & Man., 36(6):779–808 (Part 1); 809–840 (Part 2), 2000.
12. van Rijsbergen C.J. A theoretical basis for the use of co-occurrence data in information retrieval. J. Doc., 33, (2):106–119, 1977.
13. Voorhees E. Query expansion using lexical-semantic relations. In Proc. 17th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1994, pp. 61–69.
14. Xu J. and Croft B. Query expansion using local and global document analysis. In Proc. 19th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1996, pp. 4–11.

Query Expansion Models

Ben He
University of Glasgow, Glasgow, UK

Synonyms

Term expansion models

Definition

In information retrieval, the query expansion models are the techniques, algorithms or methodologies that reformulate the original query by adding new terms into the query, in order to achieve a better retrieval effectiveness.

Historical Background

The idea of expanding a query to achieve better retrieval performance emerged around the early 1970's. A classical query expansion algorithm is Rocchio's relevance feedback technique proposed in 1971 [10] for the Smart retrieval system. Since then, many different query expansion techniques and algorithms have been proposed.

Foundations

Query expansion models can be classed into three categories: manual, automatic, and interactive. Manual query expansion relies on searcher's knowledge and experience in selecting appropriate terms to add to the query. Automatic query expansion weights candidate terms for expansion by processing the documents returned from the first-pass retrieval, and expands the original query accordingly. Interactive query expansion automates the term weighting process, but it is the user who decides which are the expanded query terms.

Manual query expansion inspires and motivates the user to refine the initial query through heuristics, for instance, by providing a list of candidate terms mined from query log, and allowing user the to choose appropriate terms to add.

Automatic query expansion, different from manual query expansion, expands search queries without any form of human interaction. According to [15], the reason for not involving human interaction could be that the searcher does not want to make an effort perhaps he/she simply does not understand and does not bother adding more terms to the query. Efthimiadis [4] categorized the automatic query expansion methods into three subgroups on which the expansion process is based: search results, collection dependent data structures (e.g., term co-occurrence, term frequency distribution etc.), and collection independent data structures (e.g., lexicon relation between terms, synonyms etc.).

The first category (automatic query expansion based on search results) uses the returned documents from the first-pass retrieval with or without relevance information, as feedback for query expansion. In [10], Rocchio proposed a classical query expansion algorithm based on the Vector Space model. His algorithm has the following steps:

1. The first-pass retrieval consists of ranking the documents for the given query.
2. A term weight w(t, d) is assigned to each term occurring in one of the top-ranked document set D_{psd}. Such a document set is usually called a *pseudo relevance set*. A term weight is first assigned to each term document pair, that is the same weight assigned in the first-pass retrieval. The weighting model used is tf·idf (see TF IDF model).
3. Add the most weighted terms in the pseudo relevance set to the query, and modify the query term weights by taking into account both the original-query term weight (qtw) used in the first-pass retrieval, and the weight assigned by the term weighting model. The query used in the first-pass-retrieval is called the *original query*. Moreover, the query with the modified query term weights is called the *reweighed query*. The reweighed query-with the added query terms is called the *expanded query*. Using Rocchio's method, the new query term weight $qtw_m$is given by Rocchio's query expansion formula as follows.

$$qtw_m = \alpha \cdot qtw + \beta \cdot \sum_{d \in D_{psd}} \frac{w(t,d)}{|D_{psd}|} \quad (1)$$

If an expanded query term is not in the original query, *qtw* is zero. α and β are free parameters. The new query term weight is given by an interpolation of the original query term weight and the average term weight in the pseudo relevance set with $\alpha + \beta = 1$.

Another popular and successful automatic query expansion algorithm was proposed by Robertson [7,8] in the development of the Okapi system. Okapi's query expansion algorithm is similar to Rocchio's, while using a different term weighting function. It takes the top R documents returned from the first-pass retrieval as the pseudo relevance set. Unique terms in this set are ranked in descending order of the Robertson Selection Value (RSV) weights [7]. A number of top-ranked terms, including a fixed number of non-original query terms, are then added to the query. A major difference between Rocchio's and Robertson's methods is that the former explicitly uses relevant documents for feedback, while the latter assumes a pseudo relevance set. Okapi's query expansion method has been shown to be very effective for ad-hoc information retrieval. For example, in TREC-3 ad-hoc retrieval task, a 15.43% improvement over the Okapi BM25 baseline brought by Okapi's query expansion was reported [9].

Recently, Amati proposed a query expansion algorithm in his Divergence from Randomness (DFR) framework [1,2], which similarly follows the steps in Rocchio's algorithm. However, in Amati's method, term weights are assigned by a DFR term weighting model. Two DFR term weighting models have been proposed, namely Bo1 based on Bose-Einstein statistics, and KL based on Kullback-Leibler divergence. For example, using the Bo1 model, the weight of a term *t* in the pseudo relevance document set D(Rel) is given as:

$$w(t) = tf_x \cdot \log_2 \frac{1+\lambda}{\lambda} + \log_2(1+\lambda) \qquad (2)$$

where λ is the mean of the assumed Poisson distribution of the term in the pseudo relevant document set D(Rel). It is given by $\frac{tf_{rel}}{N_{rel}}$. tf_{rel} is the frequency of the term in the pseudo relevant documents, and N_{rel} is the number of pseudo relevant documents. tf_x is the frequency of the query term in the pseudo relevance document set.

Once the first-pass retrieval is finished, using a document weighting model, a weight is assigned to each term in the top-ranked documents returned from the first-pass retrieval. This corresponds to the first and second steps of the relevance feedback process as introduced above.

In the next step, the original query terms are reweighed. The modified query term weight qtw_m is given by the following parameter-free formula [1]:

$$qtw_m = qtw + \frac{w(t)}{M} \qquad (3)$$

where *qtw* is the original query term weight. $w(t)$ is the weight of the query term that is given by Bo1. *M* is the upper bound of the weight of a term in the top-ranked documents. For Bo1, it is computed by:

$$\begin{aligned} M &= \lim_{tf_c \to tf_{c,max}} w(t) \\ &= tf_{c,max} \log_2 \frac{1+P_{max}}{P_{max}} + \log_2(1+P_{max}) \end{aligned} \qquad (4)$$

where tf_c is the frequency of the query term in the whole collection, and $tf_{c,max}$ is the frequency of the query term with the highest $w(t)$ weight in the top-ranked documents. P_{max} is given by $tf_{c,max}/N$. *N* is the number of documents in the collection. An obvious advantage of Amati's approach is that the query term reweighting formula is parameter free. The parameter α in Rocchio's formula (see Equation (1)) is omitted.

In addition to automatic query expansion based on search results, various query expansion methods have been proposed based on collection dependent data structures (e.g., [6,11,12]), or collection independent data structures (e.g., [3,5,13]).

In interactive query expansion, on one hand, the search system offers the user a list of terms for expansion. On the other hand, it relies on the user to choose the appropriate expanded query terms. Similarly to automatic query expansion, interactive query expansion methods can also be categorized into three subgroups by on which the expansion process is based [4]: search results, collection dependent data structures, and collection independent data structures.

Key Applications

Query expansion models are employed in many information search systems such as library search systems and Web search engines for boosting their retrieval effectiveness.

Experimental Results

Query expansion models are usually helpful in improving retrieval effectiveness for general search tasks such as ad-hoc retrieval [14]. For example, a 15.43% improvement over the BM25 baseline was reported in the TREC-3 adhoc task using Okapi's query expansion method [9].

Table 1 demonstrates the effectiveness of query expansion for ad-hoc retrieval on the TREC (http://trec.nist.gov/) test data. The weighting model used for retrieval is DFRee proposed by G. Amati and implemented in the Terrier Platform (http://ir.dcs.gla.ac.uk/terrier/). Terrier's default query expansion setting is applied, which expands the original query with the 10 most informative terms from the top-3 returned documents. Table 1 shows markable improvement in the retrieval performance, measured by mean average precision, over the baseline. In all cases, the improvement is statistically significant according to the Wilcoxon matched-pairs signed-ranks test.

However, the effectiveness of query expansion becomes unreliable when there are only very few relevant documents, or when the information needed is very specific, in which cases expanding the query does

Query Expansion Models. Table 1. The mean average precision (MAP) obtained with and without query expansion

Task	MAP, No QE	MAP QE	Difference (%)	p-value
TREC-1 ad-hoc	0.2148	0.2478	15.36	4.665e-06
TREC-2 ad-hoc	0.1821	0.2370	30.15	5.306e-08
TREC-3 ad-hoc	0.2557	0.3070	20.06	2.818e-07
TERC-8 small-web	0.2829	0.3164	11.84	9.261e-05
TREC2004 robust	0.2485	0.2920	17.50	2.41e-20
TREC-9 Web	0.2034	0.2180	7.18	0.01561
TREC-10 Web	0.2027	0.2526	24.62	1.715e-06
TREC-2004 Terabyte	0.2646	0.3027	14.40	4.537e-05
TREC-2005 Terabyte	0.3293	0.3828	16.25	2.818e-07
TREC-2006 Terabyte	0.2859	0.3348	17.10	1.582e-05
TREC-2004 Genomics	0.2921	0.3398	16.33	0.0005686
TREC-2005 Genomics	0.2172	0.2500	15.10	0.001759

not bring up more relevant documents. There are also other factors that may affect the effectiveness of query expansion, such as the quality of the top-ranked documents, how much noise the document collection contains etc. For example, query expansion does not improve retrieval performance on the Blog06 test collection (http://ir.dcs.gla.ac.uk/test_collections/blog 06info.html), possibly due to the large amount of spam.

Cross-references

▶ Binary Independence Model
▶ Probabilistic Models
▶ Relevance feedback
▶ Relevance Feedback for Content-Based Information Retrieval
▶ Rocchio's Formula
▶ Web search Relevance Feedback

Recommended Reading

1. Amati G. Probabilistic models for information retrieval based on divergence from randomness. Ph.D thesis, Department of Computing Science, University of Glasgow, Glasgow, UK, 2003.
2. Carpineto C., de Mori R., Romano G., and Bigi B. An information-theoretic approach to automatic query expansion. ACM Trans. Inf. Syst., 19(1):1–27, 2001.
3. Croft B. and Das R. Experiments with query acquisition and use in document retrieval systems. In Proc. 12th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1989, pp. 349–368.
4. Efthimiadis N.E. Query expansion. Annu. Rev. Inf. Syst. Technol., 31:121–183, 1996.
5. Jarvelin K., Kristensen J., Niemi T., Sormunen E., and Keskustalo H. A deductive data model for query expansion. Technical report, Department of Information Studies, 1995.
6. Minker J., Wilson G., and Zimmerman B. An evaluation of query expansion by the addition of clustered terms for a document retrieval system. Inf. Storage Retrieval, 8:329–348, 1972.
7. Robertson S.E. On term selection for query expansion. J. Doc., 46:359–364, 1990.
8. Robertson S.E., Walker S., Beaulieu M.M., Gatford M., and Payne A. Okapi at TREC-4. In Proc. The 4th Text Retrieval Conference, 1995.
9. Robertson S.E., Walker S., Jones S., Hancock-Beaulieu M.M., and Gatford M. Okapi at TREC-3. In Proc. The 3rd Text Retrieval Conference, 1994.
10. Rocchio J. Relevance Feedback in Information Retrieval. Prentice-Hall, USA, 1971, pp. 313–323.
11. Sparck J.K. Automatic Keyword Classification for Information Retrieval. Butterworths, London, 1971.
12. Sparck J.K. Collection properties influencing automatic term classification. Inf. Storage Retrieval, 9:499–513, 1973.
13. Voorhees E. Query expansion using lexical-semantic relations. In Proc. 17th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1994, pp. 61–69.
14. Voorhees E. TREC: Experiment and Evaluation in Information Retrieval. MIT, Cambridge, MA, USA, 2005.
15. Walker S. The Okapi Online Catalogue Research Projects. Library Association, London, 1989.

Query Language

Tore Risch
Uppsala University, Uppsala, Sweden

Synonyms

Data manipulation language

Definition

A query language is a specialized programming language for searching and changing the contents of a database. Even though the term originally refers to a sublanguage for only searching (querying) the contents of a database, modern query languages such as SQL are general languages for interacting with the DBMS, including statements for defining and changing the database schema, populating the contents of the database, searching the contents of the database, updating the contents of the database, defining integrity constraints over the database, defining stored procedures, defining authorization rules, defining triggers, etc. The data definition statements of a query language provide primitives for defining and changing the database schema, while data manipulation statements allow populating, querying, as well as updating the database. Queries are usually expressed declaratively without side effects using logical conditions. However, modern query languages also provide general programming language capabilities through the definition of stored procedures. Most query languages are textual, meaning that the queries are expressed as text string processed by the DBMS. There are also graphical query languages, such as Query-By-Example (QBE), where the queries are expressed graphically and then translated into textual queries interpreted by the DBMS.

Key Points

Since data is represented in terms of the data model used by the DBMS, the syntax and semantics of a query language depends on the data model of the DBMS. For example, the relational data model is based on representing data as tables which allows expressing queries over the tables using expressions expressed in variant of predicate logic called relational calculus. Such queries are non-procedural since the user need not specify the details how a query is executed, but rather only how to select and match data from the tables. It is up to the query optimizer to translate the non-procedural logical queries into an efficient program for searching the database. Queries can also be expressed as relational algebra expressions. A query language is said to be relationally complete if its power is equivalent to the power of the relational algebra or the relational calculus. The most widespread query language is SQL the standard language used for interacting with relational DBMSs. Queries in SQL are mainly expressed as syntactically sugared relational calculus expressions. However SQL also has query constructs based on the relation algebra and other paradigms. The logical query language Datalog, as well as SQL's predecessor QUEL, is purely based on predicate calculus. Other data models use other query languages. For example, OQL is used for searching object-oriented databases and Daplex for functional databases.

Cross-references

- ▶ Data Model
- ▶ Datalog
- ▶ Daplex
- ▶ OQL
- ▶ QBE
- ▶ Relational Calculus
- ▶ SQL
- ▶ Stored Procedure

Query Languages and Evaluation Techniques for Biological Sequence Data

Sandeep Tata[1], Jignesh M. Patel[2]
[1]IBM Almaden Research Center, San Jose, CA, USA
[2]University of Wisconsin-Madison, Madison, WI, USA

Synonyms

Querying DNA sequences; Querying protein sequences

Definition

A common type of data that is used in life science applications is biological sequence data. Data such as DNA sequence and protein sequence data are growing at a very fast rate. For example, the data at GenBank [GB07] has been growing exponentially, doubling roughly every 18 months. These sequence datasets are often queried in complex ways and the methods required to query these sequences go far beyond the simple string matching methods that have been used in more traditional string applications. In order to enable users to easily pose sophisticated queries on these biological sequences, different languages have been designed to support a rich library of functions. In addition, some database systems have been extended to support a rich set of operators on the sequence data type. Compared to the stand-alone approach, the database method brings the power of algebraic query optimization and the use of indexes making it possible to find efficient execution plans for sophisticated queries. Furthermore, biological sequence processing can be integrated with traditional database processing, such as selecting a subset of data for analysis, or combining data from multiple sources, to produce a powerful sequence analysis and mining system.

Historical Background

Several research efforts have tackled the problem of enabling complex and efficient querying on biological data. Early efforts such as [4] investigated the idea of a Genomics Algebra, which would abstract several biological processes and enable users to construct complex expressions, thereby providing a sophisticated platform for processing biological sequences. As part of another effort, the Periscope project, an algebra for biological sequences called PiQA (Protein Query Algebra) was proposed in [12]. PiQA can be used to construct complex expressions that allow sophisticated manipulation of both genetic and protein sequence data. The Periscope/SQ [13] system and the PQL query language are based on this algebra.

One of the primary operations in biological sequence processing is local alignment. A dynamic programming algorithm was proposed in [9] for this problem. The popular BLAST algorithm [1] is a heuristic version proposed in 1990 to support local alignment search on massive datasets.

Foundations

The amount of biological sequence data available to scientists for analysis has grown rapidly. While small sequences can be analyzed and queried fairly quickly on computers, large datasets require careful design of algorithms and data structures. Database systems have dealt with the problem of managing and processing large datasets for several years, and the area of query processing for biological sequences aims to bring the benefits of this research to the domain of biological sequences.

One of the most common computations in the context of sequence data is that of finding "similar" sequences. This is often the first step biologists perform to begin investigation of a particular biological sequence they have discovered. For instance, finding a known protein with a sequence similar to the one under investigation may yield some clues to the function of the protein. Sequence similarity is defined in many ways and the type of similarity used may depend on the sequence and the application for which the similarity is being computed. Some distance measures are based on the evolutionary distance between the sequences, while in some cases they simply count the number of mismatches between an alignment of two sequences of the same length. Examples of evolutionary distance matrices include the PAM [3] and BLOSUM [5] matrices for comparing protein sequences. Simple models for mismatches include simply finding sequences that have up to a specified number of mismatches, or models in which mismatches are only tolerated in specific positions (such at the middle portion of the sequence).

The problem of finding a local alignment for two sequences requires finding a region of maximum similarity in a longer sequence that will match with a shorter query sequence. This version of sequence similarity is extremely popular, especially for large DNA sequence repositories. While the Smith-Waterman algorithm [9] is an elegant dynamic programming technique to compute an optimal alignment with $O(n^2)$ cost, for large sequence databases this cost can be prohibitive. In practice, methods that approximate Smith-Waterman but are much faster are used. Popular methods in this category include BLAST [1] and FASTA [8]. While these methods allow the fast evaluation of simple sequence queries, for sophisticated queries, they do not address sophisticated queris, such as a complex string pattern query. Additional methods are required that include a language to express such quires and algorithms to efficiently evaluate the operations in these queries.

Query languages provide a convenient way to express common computational tasks. The class of query languages can be divided into two categories: (i) procedural and (ii) declarative. Several procedural query languages are often general purpose programming languages or scripting languages with a library of functions that support operations on biological sequences. BioPerl and BioPython are two such examples. Figure 1 shows an example of a BioPerl code snippet that uses library function to translate the DNA sequence into the corresponding protein sequence, and another function to compute the reverse complement. These libraries also provide functions to read and translate between several popular sequence file formats. Although such languages make it somewhat easier to develop biological sequence processing applications, they suffer

```
use Bio::PrimarySeq;
my $str = "GATTACATACAT";
my $pseq = Bio::PrimarySeq->new(-seq => $str,
        -display_id=>"testseq");
print $pseq->seq, "\n";
print $pseq->translate->seq,"\n";
print $pseq->revcom->seq;
```

Query Languages and Evaluation Techniques for Biological Sequence Data. Figure 1. Sample BioPerl code.

from two major drawbacks: (i) inability to rapidly express sophisticated queries (ii) lack of an indexing and optimization infrastructure to choose efficient query plans. Applications written in procedural languages often need to solve several database problems in order to scale to larger data sets. As a result, researchers have recently focused on declarative query processing for biological data.

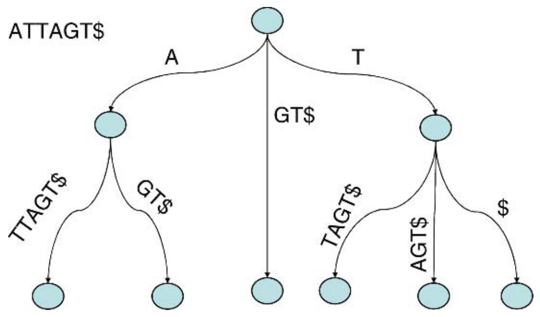
In addition to an algebra, a declarative query infrastructure needs efficient physical operators corresponding to the operators in the algebra in order to produce query evaluation plans. The research prototype system Periscope [14] uses the PiQA algebra and the PiQL query language. Figure 2 shows an example PQL query that finds all instances of the sequence "ACAC" followed by "TTACAGGG" within 100 symbols in the given sequence database. The Periscope system is built as an extension to an object-relational system, which allows the user to mix sequence queries with traditional relation data manipulation queries such as selections and joins. Relational database systems are now commonly used to manage large biological datasets (for example the GMOD project). Arguably, the declarative query processing with the relational frameworks provides a better framework (compared to the procedural framework) for complex biological data analysis.

Index Structures

To speed up the evaluation of complex biological sequence operations, a number of index-based methods have been designed. Exploiting these indexes is crucial in having efficient query execution plans, especially to cope with the increasing data volumes and query complexity. The suffix tree is one of the useful data structures in the world of string processing [15]. A Suffix tree is a tree that is built using an input string such that every path from the root of the tree to a leaf node corresponds to a suffix in the string. The edges are labeled

```
SELECT * FROM AUGMENT(
MATCH(T1, seqid, sequence, 0, "ACAC"),
MATCH(T1, seqid, sequence, 1, "TTACAGGG"), 0, 100)
```

Query Languages and Evaluation Techniques for Biological Sequence Data. Figure 2. Sample PiQL query.



Query Languages and Evaluation Techniques for Biological Sequence Data. Figure 3. Sample suffix tree built on the string "ATTAGT."

with substrings from the string. A suffix tree can be used to evaluate exact substring queries in time proportional to the length of the query. Figure 3 shows an example suffix tree on the string ATTAGT$. In order to check if the string TAG is a substring of the database string (on which the suffix tree is built), one simply follows the labeled edges from the root of the tree while trying to consume the symbols (T,A,G) from the query string.

Suffix trees can be constructed in time proportional to the length of the input string, i.e., in time O(n). Several algorithms have been designed to accomodate very large input datasets and construct disk based suffix trees. Suffix trees can also be used to efficiently answer approximate matching queries and even compute the local alignment of a query string with the database string.

Another interesting approach to indexing sequence data in the context of similarity search is the use of metric space indexing [7]. While extremely efficient for distance measures that are metrics, such indexing techniques tend to be less flexible than suffix trees.

Given the popularity of BLAST for sequence query processing, a number of approaches have added support for invoking BLAST from a database engine [2,6,11], and regular expression engines in SQL engines have also been adapted for more sophisticated biological pattern matching [10]. Their support for scanning sequences with complex patterns (including arbitrary position-specific weight matrices containing probabilistic models for matching a nucleic or protein sequence) is limited. However, such extensions have recently been made for the declarative framework using suffix trees [12].

Key Applications

Any application that requires complex sequence matching, which include combinity sequence homology matching, and TFBS prediction.

Cross-references

▶ Biological Sequences
▶ Data Types in Scientific Data Management
▶ Index Structures for Biological Sequences
▶ Query Languages for the Life Sciences
▶ Scientific Databases
▶ Suffix Tree

Recommended Reading

1. Altschul S.F., Gish W., Miller W., Myers E.W., and Lipman D.J Basic local alignment search tool. J. Mol. Biol., 215:403–10, 1990.
2. Barbara A. and Eckman A.K. Querying BLAST within a data federation. Q. Bull. IEEE TC on Data Engineering, 27(3):12–19, 2004.
3. Dayhoff M.O., Schwartz R.M., and Orcutt B.C. A model of evolutionary change in proteins. Atlas of Protein Sequence and Structure, 5:345–352, 1978.
4. Hammer J. and Schneider M. Genomics algebra: a new, integrating data model, language, and tool for processing and querying genomic information. In Proc. 1st Biennial Conf. on Innovative Data Systems Research, 2003, 176–187.
5. Henikoff S. and Henikoff J. Amino acid substitution matrices from protein blocks. In Proc. Natl. Acad. Sci., 89(22): 10915–10919, 1992.
6. Hsiao R-L., Stott Parker D., and Yang H-C. Support for BioIndexing in BLASTgres. In Data Integration in the Life Sciences (DILS). LNCS, Vol. 3615. Springer, Berlin, 2005, pp. 284–287.
7. Mao R., Xu Weijia., Singh Neha., and Miranker D.P. An assessment of a metric space database index to support sequence homology. In Proc. IEEE 3rd Int. Symp. on Bioinformatics and Bioengineering, 2003, pp. 375–382.
8. Pearson W.R. and Lipman D.J. Improved tools for biological sequence comparison. In Proc Natl Acad Sci., 85(8):2444–2448, 1988.
9. Smith T.F. and Waterman M.S. Identification of Common Molecular Subsequences. J. Mol. Biol., 147:195–197, 1981.
10. Stephens S., Chen J.Y., Davidson M.G., Thomas S., and Trute B.M. Oracle database 10 g: a platform for BLAST search and regular expression pattern matching in life sciences. Nucleic Acids Res., 33(Database-Issue):675–679, 2005.
11. Stephens S., Chen J.Y., and Thomas Shiby. ODM BLAST: sequence homology search in the RDBMS. Q. Bull. IEEE TC on Data Engineering, 27(3):20–23, 2004.
12. Tata S., Lang W., and Patel J.M. Periscope/SQ: interactive exploration of biological sequence databases. In Proc. 33rd Int. Conf. on Very Large Data Bases, 2007, pp. 1406–1409.
13. Tata S. and Patel J.M. PiQA: an algebra for querying protein data sets. In Proc. 15th Int. Conf. on Scientific and Statistical Database Management, 2003, pp. 141–150.
14. Tata S., Patel J.M., Friedman J.S., and Swaroop A. Declarative querying for biological sequences. In Proc. 22nd Int. Conf. on Data Engineering, 2006, p. 87.
15. Weiner P. Linear pattern matching algorithm. In Proceedings of the 14th Annual IEEE Symp. on Switching and Automata Theory, 1973, pp. 1–11.

Query Languages for the Life Sciences

ZOÉ LACROIX
Arizona State University, Tempe, AZ, USA

Synonyms

Biological query Languages; Scientific query Languages; Biological data retrieval, integration, and transformation.

Definition

A *scientific query language* is a query language that expresses the data retrieval, analysis, and transformation tasks involved in the dataflow pertaining to a scientific protocol (or equivalently workflow, dataflow, pipeline). Scientific query languages typically extend traditional database query languages and offer a variety of operators expressing scientific tasks such as ranking, clustering, and comparing in addition to operators specific to a category of scientific objects (e.g., biological sequences).

Historical Background

A scientific query may involve data retrieval tasks from multiple heterogeneous resources and perform a variety of analysis, transformation, and publication tasks. Existing approaches used by scientists include hard coded scripts, data warehouses, link-based federations, database mediation systems, and workflow systems. Hard coded scripts written in Perl and Python are widely used in the biological community. Unlike an approach based on a query language, scripts are very limited in terms of scalability and flexibility. Extending or altering a protocol over time requires writing new scripts. Further, scripts are limited in the degree to which they capture biological expertise so they can be re-used for future related queries. Finally, unless explicitly written to do so, scripts do not assist the user in filtering retrieved data, resolving inconsistencies and sorting and ranking the results, or optimizing the overall execution. Data warehousing consists in collecting data from many possible data sources, curating the data, and creating a new database. Data warehouses typically are relational databases designed to provide users an integrated platform to answer a pre-defined set of scientific tasks.

A federation, e.g., NCBI Entrez and the Sequence Retrieval System (SRS) [4], links semi-autonomous distributed databases with powerful full text indexing and keyword based search techniques for cross database retrieval. The approach expresses navigational queries over linked entries retrieved from multiple databases. In addition, useful tools such as BLAST are often made available. However, the approach is limited in that a federation relies on a materialized index that is difficult to keep up-to-date. Moreover, it focuses on data retrieval and does not support complex queries. Most existing data integration approaches rely on an internal query language that captures data management operations and a user query interface that aims at expressing queries meaningful to the scientists. Existing mediation approaches rely on traditional database query languages such as SQL adapted to handle biological data. K2 follows the Object Data Management Group (ODMG 1999) and its internal query language, called K2 Mediator Definition Language (K2MDL), is a combination of the Object Query Language (OQL) and the Object Definition Language (ODL) both specified by the ODMG. Kleisli (also known as Discovery Hub) provides the collection programming language (CPL) [2], a nested version of SQL, and Java and Perl access programming interfaces (API). Users may express their queries in SQL or through a graphical interface that limits access to query capabilities. The system that supports the Object Protocol Model (OPM) [3] provides a query language similar to OQL (ODMG 1999) and also exploits SQL when integrated data sources are retrofitted from a relational data model. Additional query capabilities may be integrated through CORBA classes. Tools such as BLAST are wrapped through an Application Specific Data Type (ASDT). Users may express their queries in a simplified version of OQL and generate query forms that do not offer access to all query capabilities. Discovery Link (also known as Information Integrator) offers a rich subset of SQL3 (SQL 1993) including user-defined functions, stored procedures, recursion, row types, object views, etc. P/FDM provides support to access specific capabilities of sources. P/FDM uses Daplex that offers a richer syntax than SQL [10] and Prolog. In particular it allows uses of function calls within queries. In addition to these query languages, P/FDM offers users a visual interface that enables users to build their queries by browsing and clicking through the database schema. The P/FDM Web interface uses HTML forms and access the mediator through CGI programs. Both interfaces restrict the query capabilities of the mediator. TAMBIS, which uses CPL internally, is primarily concerned with overcoming semantic heterogeneity through the use of ontologies. It provides users an ontology-driven browsing interface. Thus it too restricts the extent to which sources can be exploited for scientific discovery. To summarize, these systems have made many inroads into the task of data integration from diverse data sources. However they all rely on significant programming effort to adjust to specific scientific tasks, are difficult to maintain and provide user's query language that require programming ability such as SQL, OQL, Daplex, etc. or user's friendly interfaces that significantly limit the query capabilities. A comparative study of these systems and the many causes of failure of traditional database approaches to support the process of scientific discovery are detailed in [6].

Workflows are used in business applications to assess, analyze, model, define, and implement business processes. A workflow automates the business procedures where documents, information or tasks are passed between participants according to a defined set of rules to support an overall goal. In the context of scientific applications, a workflow approach may address the collaboration among scientists, as well as the integration of scientific data and tools. The procedural support of a workflow resembles the query-driven design of scientific problems and facilitates the expression of scientific pipelines (as opposed to a database query). However, because workflows are designed to orchestrate various applications (e.g., Web services) into a combined dataflow they do not provide a query language and do not express the specific queries against biological datasets. Peer-to-Peer (e.g., Chinook) and grid approaches (e.g., myGRID) may offer the support for resource and data sharing without providing a query language.

Foundations

Systems that focus on specific datasets such as sequences, protein structure, phylogenetic trees, metabolic pathways, rely on query languages designed to handle the characteristics of the biological datasets. Life sciences data are exceptionnaly diverse. Biological data include string data (e.g., sequences over various alphabets), 3D geometric data (e.g., protein structure), tree data (e.g., phylogenetic tree), and graph data (e.g., pathways). When their description and annotations are mostly textual and deeply nested in one or several documents, the intrinsic structure of the datasets are often poorly represented by traditional data models, thus poorly accessed and transformed by the corresponding query languages [7]. Each of these specific biological datasets has motivated the design of suitable query languages. For example PiQA, an algebra designed to query sequences, offers a match operator that expresses sequence alignment.

Query languages that express search queries (using regular expressions, wildcards, and Boolean operators) are often developed for life scientific applications. This is because scientific data are mostly textual (annotation on scientific instances) and scientists seem to mimic the manual query process they follow when exploring data sources on the Web. Querying systems that are compatible with scripting languages such as Perl are also favored because most scientific protocols are implemented with scripts. Additional features expected by scientists include the ability to compare and rank results. The ability to express cross-database queries and to integrate scientific analysis tools such as BLAST has a significant value to the scientist.

Scientific queries may exploit metadata as well as data. Querying scientific data through their meaning expressed in a terminology or an ontology is scientist-friendly because it hides the complex structure of data as they are organized and stored in the various repositories. A domain ontology may also provide a global schema for multiple integrated scientific resources. It facilitates scientific tasks such as *profiling* which aims at combining all information known about a scientific object. TAMBIS is an example of a system that provides a ontology-driven query interface.

The challenge for the design of a biological query language is to express the data management tasks involved in the scientific dataflow while offering meaningful and scientist-friendly query functionality. A scientific query language typically handles complex datatypes including text, string, tree, graph, list, variant, etc. for data and metadata. Scientific queries may invoke an increadibly diverse range of functions including extracting information in a textual document, identifying a feature common to two biological sequences, exploring a biological pathway, profiling a gene, etc.

Key Applications

The Acedb Query Language (AQL) is the language developed for the AceDB genome database. It is inspired by the Object Query Language (OQL), and Lorel the query language developed for semi-structured and XML data (also based on OQL). AQL expressions use the Select, From, Where syntax. Queries in AQL express traditional database retrieval and transformation against a data warehouse mostly populated with annotated scientific data (textual). It provides AcePerl and object-oriented Perl module access local or remote AceDB databases transparently. Many biological resources use the AceDB system that is still updated and maintained by the Sanger institute. The former version of AQL consisting of search statements of the form ”find Gene TP*” is currently used to query WormBase.

SRS has its own query language that express retrieval queries using string comparison, wildcards, regular expresions, numeric comparison, Boolean operations, and the *link* operator specific to SRS that allows the navigation through resources exploiting the interal cross-references (hyperlinks, indices, and composite structures) made available by the providers. SRS uses Icarus as its internal interpreted object-oriented programming language. SRS queries return sets of entries or lists of entry identifiers that can be sorted with respect to various criteria. Because SRS integrates scientific analysis tools (e.g., BLAST) as tool-specific databanks, SRS queries may exploit the expressive power of the scientific analysis tools integrated in the federation. However, the SRS query language mostly expresses retrieval queries over cross databases. For a detailed description of the system and its query language see Chapter 5 in [6]. SRS is currently updated and mainted by Biowisdom Ltd.

The Life Sciences community has developed unique approaches to handle complex retrieval and integration tasks. They express the query “retrieve everything known about [a specific scientific instance].” Unlike a query language, the resources where the data are to be retrieved and the criteria for retrieval cannot be selected by the user. These approaches include sequence profiling tools that express complex data retrieval and integration queries over multiple heterogeneous resources to generate summaries of all information related to a particular biological sequence. For example, the Karlsruhe Bioinformatik Harvester (KIT) crawls and crosslinks 25 biological resources. Data integration is limited as queries against KIT are species-specific boolean expressions of keywords and the result is a list of summaries (each summary corresponds to a biological sequences), where each summary is the concatenation of relevant retrieved pages. Other approaches focus on the textual material contained in the biological data sources and use biological language processing to achieve information retrieval, extraction, classification, and integration [5]. These approaches retrieve documents from various resources and generate a document summary as input. Textpresso uses the Gene Ontology (GO) to markup documents related to a particular organism retrieved from PubMed. It provides a query language that expresses Boolean expressions on keyword, category, and attribute. Because the approach is document-driven, the query language expresses advanced search queries. Patent databases are other documented sources of biological information. In addition to the traditional features of querying documented data, Patent Lens offers the useful feature of querying the biological sequences contained in the patent documents with BLAST [9].

The use of a domain ontology as an interface (or view) for biological data is usually well received by the scientists. Such approches typically support exploration, navigation, and search queries. TAMBIS uses an ontology to provide an homogeneous layer over the integrated resources (data sources and applications) that acts as a global schema, to hide the heterogeneities of the integrated resources, and to be used as a query interface. WormBase is a warehouse derived from ACeDB and BioMart databases and uses biomaRt, a bioconductor package that provides a platform for data mining. BioMart databases are annotated with the Gene Ontology (GO) and the eVOC ontologies. These annotations provide support for querying data. EnsMart organizes data with respect to *foci*, that are central scientific objects (e.g., gene). All data are linked to the instances of those central objects. Queries consist of the selection of the species and focus of interest and the specification of filters and outputs. AmiGO provides a query interface to GO, genes, and gene products.

The Pathway Tools developed and maintained at SRI International to support pathway databases produces a query form for basic or advanced search against BioCyc databases. Each form consists of the selection of the BioCyc database (queries are limited to the scope of a single resource), the field(s) to be searched or the ontology term(s), and the format of the output. Most of the databases developed with BioCyc allow the use of BLAST. The BioVelo query language is designed to retrieve data from BioCyc databases. It is based on the object-oriented BioCyc data model composed of object classes that are identified to ontology classes and express set comprehension statements used in functional programming languages such as Python.

Future Directions

The design of query languages for the Life Sciences is a challenging problem. The desired expressive power and features of the languages are not well understood. The expectations of the users (life scientists) and the developers (computer scientists) are often dissimilar. Life scientists expect support for all their scientific questions which often go beyond the scope of traditional database query languages handling data retrieval and transformation. Scientific questions are often complex protocols that invoke not only data retrieval and transformation tasks but ranking, comparison, clustering and classification as well as sophisticated analysis tasks. The implementation of such a "query" would require accessing multiple resources and coordinating the dataflow as currently achieved by scientific workflow systems such as Taverna. However, workflow systems lack the elegance of a query language and its benefits such as query planning and optimization. Because many of the scientific tasks invoked in a scientific protocol could be expressed by operators of a query language, the design of a generic query language to handle data retrieval, transformation, comparison, clustering, and integration operations would be valuable to scientific data management.

Data Sets

AceDB databases http://www.acedb.org/Databases/
Public SRS servers http://downloads.biowisdomsrs.com/publicsrs.html

Biopathways Graph Data Manager (BGDM) http://hpcrd.lbl.gov/staff/olken/graphdm/graphdm.htm
BioCyc database collection http://www.biocyc.org/
Textpresso for C. elegans http://www.textpresso.org/
Patent Lens http://www.patentlens.net/daisy/patentlens/patentlens.html
WormBase http://www.wormbase.org/ and http://www.wormbase.org/db/searches/wb_query
BioMart http://www.biomart.org/
EnsMart http://www.ensembl.org/EnsMart

URL to Code

Chinook http://www.bcgsc.bc.ca/chinook/
myGRID http://www.mygrid.org.uk/
AceDB Query Language http://www.acedb.org/Cornell/aboutacedbquery.html
AQL - Acedb Query Language http://www.acedb.org/Software/whelp/AQL/
Lorel - Lore Query Language http://infolab.stanford.edu/lore/
BioVelo Query Language http://www.biocyc.org/bioveloLanguage.html
BioCyc Pathway Tools http://bioinformatics.ai.sri.com/ptools/ptools-overview.html
Karlsruhe Bioinformatik Harvester (KIT) http://harvester.fzk.de/harvester/
biomaRt and Bioconductor http://www.bioconductor.org/
AmiGO http://amigo.geneontology.org/cgi-bin/amigo/go.cgi

Cross-reference

- ▶ Database mediation
- ▶ Data integration
- ▶ Graph management for the life sciences
- ▶ Management of gene expression data
- ▶ Metadata management and resource discovery
- ▶ Scientific workflows.

Recommended Reading

1. Bartlett J. C. and Toms. E.G. Developing a Protocol for Bioinformatics Analysis: An Integrated Information Behaviors and Task Analysis Approach. J. American. Soc. for Inf. Sci. & Tech., 56(5):469–482, 2005.
2. Buneman P., Naqvi S.A., Tannen V., and Wong L. Principles of Programming with Complex Objects and Collection Types. Theoretical Computer Science, 149(1):3–48, 1995.
3. Chen and I-Min A. Markowitz. Victor M. An Overview of the Object-Protocol Model (OPM) and OPM Data Management Tools. Inf. Syst., 20(5):393–418, 1995.
4. Etzold T., Harris H., and Beaulah S. SRS: An Integration Platform for Databanks and Analysis Tools in Bioinformatics, chapter 5, pp. 109–146. In Lacroix and Critchlow [6], 2003.
5. Hunter L. and Bretonne K. Cohen. Biomedical Language Processing: Perspective What's Beyond PubMed? Molecular Cell, 21 (5):589–594, 2006.
6. Lacroix Z. and Critchlow T. (eds.) Bioinformatics: Managing Scientific Data. Morgan Kaufmann, 2003.
7. Lacroix Z., Ludaescher B., and Stevens R. Integrating Biological Databases, chapter 42, pp. 1525–1572. Vol. 3 of Lengauer [8], 2007.
8. Lengauer T. (ed). Bioinformatics - From Genomes to Therapies. Wiley-VCH Publishers, 2007
9. Seeber I. Patent searches as a complement to literature searches in the life sciences-a 'how-to' tutorial. Nature Protocols, 2(10): 2418–28, 2007.
10. Shipman D.W. The Functional Data Model and the Data Language DAPLEX. ACM Trans. Database Syst., 6(1):140–173, 1981.
11. Stevens R., Goble C., Baker P., and Brass A. A Classification of Tasks in Bioinformatics. Bioinformatics, 17(2):180–188, 2001.

Query Load Balancing in Parallel Database Systems

Luc Bouganim
INRIA, Rocquencourt, Lechesnay Cedex, France

Synonyms

Resource scheduling

Definition

The goal of parallel query execution is minimizing query response time using inter- and intra-operator parallelism. Inter-operator parallelism assigns different operators of a query execution plan to distinct (sets of) processors while intra-operator parallelism uses several processors for the execution of a single operator thanks to data partitioning. Conceptually, parallelizing a query amounts to divide the query work in small pieces or tasks assigned to different processors. The response time of a set of parallel tasks being that of the longest one, the main difficulty is to produce and execute these tasks such that the query load is evenly balanced within the processors. This is made more complex by the existence of dependencies between tasks (e.g., pipeline parallelism) and synchronizations points. Query load balancing relates to static and/or dynamic techniques and algorithms to balance the query load within the processors so that the response time is minimized.

Historical Background

Parallel database processing appeared very early in the context of database machines in the 1970s. Parallel algorithms (e.g., hash joins) were later proposed in the early 1980s, where tuples are uniformly distributed at every stage of the query execution. However several works (e.g., [8]) gave considerable evidence that data skew, i.e., non-uniform distribution of tuples, exists and its negative impact on query execution was shown in e.g., [7]. This motivated numerous studies [10] on intra- and inter-operator load balancing in the 1990s.

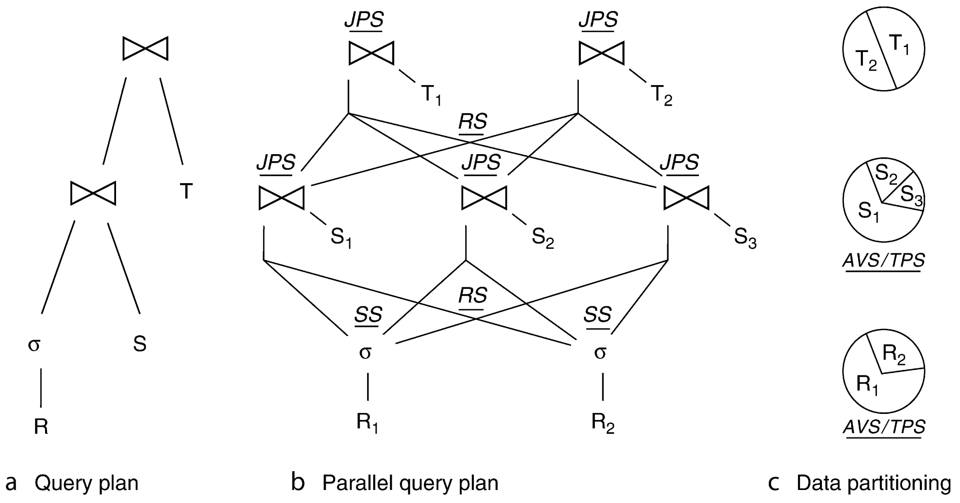
Foundations

Load balancing problems can appear with intra-operator parallelism (variation in partition size, namely *data skew*) and inter-operator parallelism (variation in the complexity of operators, synchronization problems). Intra- and inter-operator load balancing problems are first detailed on a simplified query execution plan example considering a static allocation of processors to the query operators. The main load balancing techniques proposed to address these problems are described next:

Load balancing problems

Figure 1(a) shows a simplified query execution plan for the following query: "Select T.b from R, S, T where R.Rid = S.Rid and S.Sid = T.Sid and R.a = value" (the Scan and Project operators were omitted to simplify the drawing). The following assumptions are made: (i) the degree of parallelism (i.e., number of processors allocated) for the selection on R (called σR), the join with S (called ⋈S) and the join with T (called ⋈T) has been statically fixed, using a cost model, to respectively 2, 3, and 2; and (ii) these operators are processed in pipeline, thus leading to a total degree of parallelism of 7.

Intra-operator load balancing issues are first illustrated using the classification proposed in [13]. As shown in Fig. 1(c), R and S are poorly partitioned because of *Attribute Value Skew (AVS)* inherent in the dataset and/or *Tuple Placement Skew (TPS)*. The processing time of the two instances σR1 and σR2 are thus not equal. The case of ⋈S is likely to be worse (see Fig. 1b). First, the number of tuples received can be different from one instance to another because of poor redistribution of the partitions of R (*Redistribution Skew, RS*) or variable selectivity according to the partition of R processed (*Selectivity Skew, SS*). Finally, the uneven size of S partitions (*AVS/TPS*) yields different processing times for tuples sent by the σR operator and the result size is different from one partition to the other due to join selectivity (*Join Product Skew, JPS*). The skew effects are therefore propagated toward the query tree and even with a perfect partitioning of T, the processing time of ⋈T1 and ⋈T2 can be highly different (uneven size of their left input resulting from ⋈S). Intra-operator load balancing is thus difficult to achieve statically, given the combined effects of different types of data skew.



Query Load Balancing in Parallel Database Systems. Figure 1. Intra- and inter-operator load balancing problems on a simple example.

In order to obtain good load balancing at the inter-operator level, it is necessary to choose how many and which processors to assign to the execution of each operator. This should be done while taking into account pipeline parallelism, which requires inter-operator communication and introduces precedence constraints between operators (i.e., an operator must be terminated before the next one begins). In [15], three main problems are described: (i) the degree of parallelism and the allocation of processors to operators, when decided in the parallel optimization phase, are based on a possibly inaccurate cost model. Indeed, it is difficult, if not impossible, to take into account highly dynamic parameters like interference between processors, memory contentions, and obviously, the impacts of data skew; (ii) the choice of the degree of parallelism is subject to errors because both processors and operators are discrete entities. For instance, considering Fig. 1(b), the number of processors for σR, $\bowtie$S and $\bowtie$T may have been computed by the cost model as respectively 1.5, 3.8 and 2.4 and have been rounded to 2, 3 and 2 processors. But the good distribution, taking into account data skew on S partitions should have been 1, 4 and 2; (iii) the processors associated with the latest operators in a pipeline chain may remain idle a significant time. This is called the pipeline delay problem. For instance, while tuples do not match the selection on R or the join with S, processors assigned to $\bowtie$T remain idles.

In a shared-nothing architecture, the inter-operator load balancing problem is even more complex, since the degree of parallelism and the set of processors assigned for some operators is constrained by the physical placement of the manipulated data. For instance, if R is partitioned on two nodes, σR must be executed on these nodes.

This simple example thus shows that static allocation of processors to operators is usually far from optimal, thus advocating for more dynamic strategies. In the following section, existing proposals at the intra- and inter-operator level are detailed.

Intra-Operator Load Balancing

Good intra-operator load balancing depends on the degree of parallelism and on the allocation of processors for the operator. For some algorithms, e.g., the parallel hash join algorithm, these parameters are not constrained by the placement of the data. Thus, the home of the operator (the set of processor where it is executed) must be carefully decided. The skew problem makes it hard for a parallel query optimizer to make this decision statically (at compile-time) as it would require a very accurate and detailed cost model. Therefore, the main solutions rely on adaptive techniques or specialized algorithms which can be incorporated in the query optimizer/processor. These techniques are described below in the context of parallel joins, which has received much attention. For simplicity, each operator is given a home either statically or just before execution, as decided by the query optimizer/processor.

Adaptive techniques: The main idea is to statically decide on an initial allocation of the processors to the operator (using a cost model) and, at execution time, adapt this decision to skew using load reallocation. A simple approach to load reallocation is detecting the oversized partitions and partition them again onto several processors (among those already allocated to the operator) to increase parallelism [6]. This approach is generalized in [2] to allow for more dynamic adjustment of the degree of parallelism. It uses specific *control operators* in the execution plan to detect whether the static estimates for intermediate result sizes differ from the run-time values. During execution, if the difference between estimate and real value is high enough, the control operator performs data redistribution in order to prevent join product skew and redistribution skew. Adaptive techniques are useful to improve intra-operator load balancing in all kinds of parallel architectures. However, most of the work has been done in the context of shared-nothing where the effects of load unbalance are more severe on performance.

Specialized algorithms: Parallel join algorithms can be specialized to deal with skew. The approach proposed in [3] is to use multiple join algorithms, each specialized for a different degree of skew, and to determine the best at execution time. It relies on two main techniques: range partitioning and sampling. Range partitioning is used instead of hash partitioning (in the parallel hash join algorithm) to minimize redistribution skew of the building relation. Thus, processors can get partitions of equal number of tuples, corresponding to different ranges of join attribute values. To determine the values that delineate the range values, sampling of the building relation is used to produce a histogram of the join attribute values, i.e., the numbers of tuples for each attribute value.

Sampling is also useful in determining which algorithm to use and which relation to use for building or probing. The parallel hash join algorithm can then be adapted to deal with skew as follows: (i) Sample the building relation to determine the partitioning ranges. (ii) Redistribute the building relation to the processors using the ranges. Each processor builds a hash table containing the incoming tuples. (iii) Redistribute the probing relation using the same ranges to the processors. For each tuple received, each processor probes the hash table to perform the join. This algorithm can be further improved to deal with high skew using additional techniques and different processor allocation strategies [3]. A similar approach is to modify the join algorithms by inserting a scheduling step which is in charge of redistributing the load at runtime [14].

Inter-Operator Load Balancing

The inter-operator load balancing problem was extensively addressed during the 1990s. Since then many processor allocation algorithms have been proposed for different target parallel architectures and considering CPU, I/Os or other resources, such as available memory.

The main approach in shared-nothing is to determine dynamically (just before the execution) the degree of parallelism and the localization of the processors for each operator. For instance, the Rate Match algorithm [9] uses a cost model in order to define the degree of parallelism of operators having a producer-consumer dependency such that the producing rate matches the consuming rate. It is the basis for choosing the set of processors which will be used for query execution (based on available memory, CPU, and disk utilization). Many other algorithms are possible for the choice of the number and localization of processors, for instance, by a dynamic monitoring and adjustment of the use of several resources (e.g., CPU, memory and disks) [11].

Shared-disk and shared-memory architectures provide more flexibility since all processors have equal access to the disks. Hence there is no need for physical relation partitioning and any processor can be allocated to any operator [12].

Considering the shared-disk architecture, Hsiao et al. [5] propose to assign processors recursively from the root up to the leaves of a so-called *allocation tree*. This tree is derived from the query tree, each pipeline chain (i.e., set of operators having pipeline dependencies) being represented as a node. The edges of the allocation tree represent precedence constraints. All available processors are assigned to the root node of the allocation tree (the last pipeline chain to be executed). Then, a cost model is used to divide the CPU power between each child of the root in order to ensure that all the data necessary for the execution of the root pipeline chain will be produced synchronously.

The approach proposed in [4] for shared-memory allows the parallel execution of independent pipeline chains called tasks. The main idea is combining IO-bound and CPU-bound tasks to increase system resource utilization. Before execution, a task is classified as IO-bound or CPU-bound using cost model information. CPU-bound and IO-bound tasks can then be run in parallel at their optimal IO-CPU balance, by dynamically adjusting the degree of intra-operator parallelism of the tasks.

Intra-Query Load Balancing

Intra-query load balancing combines intra- and inter-operator parallelism. To some extent, given a parallel architecture, the load balancing techniques presented above can be extended or combined. For instance, the control operators used a-priori for intra-operator load balancing can modify the degree of parallelism of an operator, thus impacting inter-operator load balancing [2].

A general load balancing solution in the context of hierarchical parallel architectures (a shared-nothing system whose nodes are shared-memory multiprocessors) is the execution model called Dynamic Processing (DP) [1]. In such systems, the load balancing problem is exacerbated because it must be addressed both locally (among the processors of each shared-memory node) and globally (among all nodes). The basic idea of DP is decomposing the query into self-contained units of sequential processing, each of which can be carried out by any processor. Intuitively, a processor can migrate horizontally (intra-operator parallelism) and vertically (inter-operator parallelism) along the query operators. This minimizes the communication overhead of inter-node load balancing by maximizing intra and inter-operator load balancing within shared-memory nodes.

Key Applications

Load balancing techniques are essential in applications dealing with very large databases and complex queries, e.g., data warehousing, data mining, business intelligence and more generally all OLAP (On Line Analytical Processing) applications.

Data Sets

DBGen, a synthetic data generator can be used to generate biased data distribution, for studying intra-operator load-balancing issues. It allows generating data with non-uniform distribution (Zipfian, Poisson, Gaussian, etc). See http://research.microsoft.com/∼Gray/DBGen/

Cross-references

- ▶ Parallel Architectures
- ▶ Parallel Database Management
- ▶ Parallel Data Placement
- ▶ Parallel Query Processing
- ▶ Storage Resource Management

Recommended Reading

1. Bouganim L., Florescu D., and Valduriez P. Dynamic load balancing in hierarchical parallel database systems. In Proc. 22th Int. Conf. on Very Large Data Bases, 1996, pp. 436–447.
2. Brunie L. and Kosch H. Control strategies for complex relational query processing in shared nothing systems. ACM SIGMOD Rec., 25(3):34–39, 1996.
3. DeWitt D.J., Naughton J.F., Schneider D.A., and Seshadri S. Practical skew handling in parallel joins. In Proc.18th Int. Conf. on Very Large Data Bases, 1992, pp. 27–40.
4. Hong W. Exploiting inter-operation parallelism in XPRS. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1992, pp. 19–28.
5. Hsiao H., Chen M.S., and Yu P.S. On parallel execution of multiple pipelined hash joins. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 185–196.
6. Kitsuregawa M. and Ogawa Y. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer. In Proc. 16th Int. Conf. on Very Large Data Bases, 1990, pp. 210–221.
7. Lakshmi M.S. and Yu P.S. Effect of skew on join performance in parallel architectures. In Int. Symp. Databases in Parallel and Distributed Systems, 1988, pp. 107–120.
8. Lynch C. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In Proc. 14th Int. Conf. on Very Large Data Bases, 1988, pp. 240–251.
9. Metha M. and DeWitt D. Managing intra-operator parallelism in parallel database systems. In Proc. 21th Int. Conf. on Very Large Data Bases, 1995, pp. 382–394.
10. Özsu T. and Valduriez P. Principles of Distributed Database Systems (2nd edn.). Prentice Hall, 1999 (3rd edn., forthcoming).
11. Rahm E. and Marek R. Dynamic multi-resource load balancing in parallel database systems. In Proc. 21th Int. Conf. on Very Large Data Bases, 1995.
12. Shekita E.J. and Young H.C. Multi-join optimization for symmetric multiprocessor. In Proc. 19th Int. Conf. on Very Large Data Bases, 1993, pp. 479–492.
13. Walton C.B., Dale A.G., and Jenevin R.M. A taxonomy and performance model of data skew effects in parallel joins. In Proc. 17th Int. Conf. on Very Large Data Bases, 1991, pp. 537–548.
14. Wolf J.L., Dias D.M., Yu P.S., Turek J. New Algorithms for parallelizing relational database joins in the presence of data skew. IEEE Trans. Knowl. Data Eng., 6(6):990–997, 1994.
15. Wilshut N., Flokstra J., and Apers P.G. Parallel evaluation of multi-join queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995, pp. 115–126.

Query Mapping

▶ Query Translation

Query Optimization

Evaggelia Pitoura
University of Ioannina, Ioannina, Greece

Synonyms

Query compilation

Definition

A query optimizer translates a query into a sequence of physical operators that can be directly carried out by the query execution engine. The output of the optimizer is called a *query execution plan.* The execution plan may be thought of as a dataflow datagram that pipes data through a graph of query operators. The goal of query optimization is to derive an efficient execution plan in terms of relevant performance measures, such as memory usage and query response time. To achieve this, the optimizer needs to provide: (i) a space of execution plans (search space), (ii) cost estimation techniques to assign a relevant cost to each plan in the search space, and (iii) an enumeration algorithm to search through the space of plans.

Key Points

The query optimizer takes as input a parsed query and produces as output an efficient execution plan for the query. The task of the optimizer is nontrivial, since given a query (i) there are many logically equivalent algebraic expressions (for instance, resulting from the commutativity property among the logical operators), and (ii) for each expression, there are many physical operators supported by the query execution engine for implementing each logical operator (for example, there are several join algorithms, e.g., nested-loop and sort-merge join, for implementing join). The task of finding an equivalent algebraic expression is often called *query rewriting*.

The optimizer needs to enumerate all possible execution plans, estimate their cost and select the one with the smallest cost. The cost assigned to each plan is based on a *cost model* that provides an estimation of the resources needed for its execution, where the resources include CPU time, I/O cost, memory, and communication bandwidth. The cost model relies on statistics maintained on relations and indexes, and uses cost formulas for estimating the selectivity of the various operators and their expected recourse usage. Often, dynamic programming techniques are used to enumerate different plans. These techniques, exploit the fact that to obtain an optimal plan for an expression, it suffices to consider only the optimal plans for its sub-expressions [1,2,3].

Cross-references

▶ Query Optimization (in Relational Databases)
▶ Query Optimization in Sensor Networks
▶ Query Plan
▶ Query Processing
▶ Query Rewriting

Recommended Reading

1. Chaudhuri S. An overview of query optimization in relational systems. In Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, 1998, pp. 34–43.
2. Ioannidis Y. Query optimization. In Handbook of Computer Science, A.B. Tucker (ed.). CRC Press, 1996.
3. Jarke M. and Koch J. Query optimization in database systems. ACM Comput. Surv., 16(2):111–152, 1984.

Query Optimization (in Relational Databases)

Thomas Neumann
Max-Planck Institute for Informatics, Saarbrücken, Germany

Synonyms

Query compilation

Definition

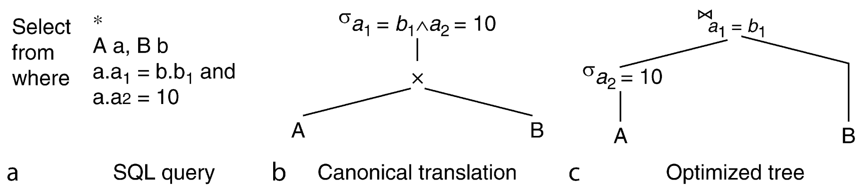
Database queries are given in declarative languages, typically SQL. The goal of query optimization is to choose the best execution strategy for a given query under the given resource constraints. While the query specifies the user intent (i.e., the desired output), it does not specify how the output should be produced. This allows for optimization decisions, and for many queries there is a wide range of possible execution strategies, which can differ greatly in their resulting performance. This renders query optimization an important step during query processing.

Historical Background

One of the first papers to discuss query optimization in relational database systems was the seminal System R paper [2]. It introduced a dynamic programming algorithm for optimizing the join order, and coined the concept of interesting orders for exploiting available orderings. Later approaches increased the set of optimized operators, and included a rule based description of optimization techniques (whereas the optimizations in System R were hard wired). One prominent example is the Starburst [3] optimizer. It introduced a different internal representation (query graph model), that could express complex queries suitably for optimization, and proposed using grammar-like rules to combine low level physical operators (LOLEPOPs) in execution plans. Besides being rule-based, the optimization itself used a bottom-up approach similar to System R. Another family of optimizers was introduced by Volcano [4] (which itself evolved from Exodus) and Cascades [5]. Instead of bottom-up constructive optimization they used transformative top-down optimization with memorization. Besides these more fundamental approaches, a rich literature of optimization techniques exist, ranging from support for specific operators like outer joins or expensive predicates to fundamental data reduction like magic sets.

Foundations

The key idea behind query optimization is the observation that the same query can be formulated in different ways. When a user gives a query in SQL, the query is first parsed and analyzed, and then brought into an internal representation, for example in relational algebra. This translation first creates a canonical representation, as shown in Fig. 1. As a first translation



Query Optimization (in Relational Databases). Figure 1. Translating a SQL query.

step, the *select-from-where* queries in a) can be answered by combining all relations in the *from* clause with a cross product and then checking the *where* condition on each resulting tuple (shown in b)). But cross products are expensive operations, therefore it is preferable to move a part of the *where* condition into the cross product to form a join. The remaining part of the condition can be checked before the join, which further reduces the effort for the join (shown in c)). Both operator's trees produce the same tuples when executed, and thus are different representations of the same query. But executing the tree in (i) is most likely cheaper than executing the tree in (ii), which means that (iii) is preferable. The query optimizer therefore starts by translating the query into a canonical representation, which is easy to construct but inefficient to execute, and therefore finds better representations of the same query.

The base for finding better alternatives is the concept of *algebraic equivalences*. Two algebra expressions are equivalent if they produce the same result when executed. As this is difficult to decide in general, query optimizers instead rely on known equivalences. For example the join operator is commutative and associative:

$$A \bowtie B \equiv B \bowtie A$$
$$A \bowtie (B \bowtie C) \equiv (A \bowtie B) \bowtie C$$

Many equivalences are known from the literature [7]. The equivalences form the search space that is explored by the query optimizer: When two (sub-)expressions are equivalent, the optimizer is free to choose any of the two. Optimizers that are directly based upon this principle are called *transformative* optimizers, as they transform algebra expressions into other algebra expressions by applying algebraic equivalences. Transformative optimizers are relatively easy to build and can potentially make use of arbitrary equivalences, but an efficient exploration of the search space is very difficult. Most transformative optimizers are therefore only heuristical. The family of *constructive* optimizers does not apply the equivalences directly, but builds expressions bottom-up from smaller expressions such that the resulting expression is still equivalent to the original expression. This allows for a much more efficient exploration of the search space, but is difficult to organize for more complex equivalences. Therefore most constructive optimizers are at least partially transformative, applying transformative rewrite heuristics for complex equivalences before (and after) the constructive optimization step.

The goal of query optimization is improving query processing, which means that the query optimizer needs to take into account the runtime effect of different alternatives. This is done by estimating the *costs* of executing an alternative. A primitive way to estimate the costs is to estimate the number of tuples processed. The intuition here is that a larger number of processed tuples implies more effort spent on executing the query. This estimation requires statistical information, in particular the cardinalities of the relations and the selectivities of the operators involved, but given these it can be computed directly from the algebra expression. Unfortunately this is much too inaccurate for practical purposes. A proper cost function should model the expected execution time (as this is the most common optimization goal), which implies taking into account access patterns on disk, costs for evaluating expensive predicates, etc. The cost function is therefore usually a linear combination of expected I/O costs and expected CPU costs. But this information cannot be derived from the algebra expression, as it is not detailed enough.

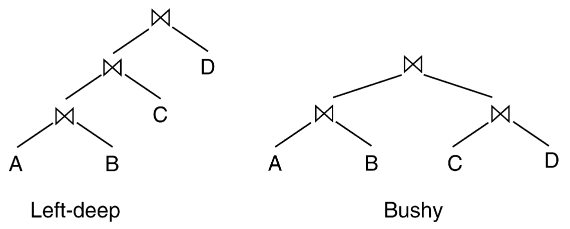
Optimizers therefore distinguish between *logical algebra* and *physical algebra*. The logical algebra consists of all operator concepts known to the optimizer, while the physical algebra consists of the operator implementations supported by the execution engine. For example the logical algebra contains one inner join operator $\bowtie$, while the physical algebra contains one join operator for each supported implementation, like nested loop join $\bowtie^{NL}$ or sort-merge join $\bowtie^{SM}$. While the initial query is represented in logical algebra (or an equivalent calculus), the final result must be in physical algebra, e.g., the optimizer must decide which operator implementations should be used. As these physical algebra expressions (including some annotations) could be executed by the runtime system, they are often called *query execution plans*. The logical algebra is more abstract, which can be useful for optimization, but ultimately the optimizer must construct physical algebra expressions. In particular, cost-based optimization requires physical algebra, as then only costs can be estimated.

Optimizing Simple Queries

The Select-Project-Join queries (SPJ) are relatively simple yet commonly used. They correspond to SQL queries of the form SELECT ... FROM ... WHERE ... without any nested queries. They can be answered by using only selections (σ), joins/cross-products ($\bowtie/\times$), and projections (Π). Nevertheless, it has been shown that finding the optimal execution plan is NP hard in general, even for these simple queries. Several simplification are commonly used to reduce the optimization time. As a first step, the optimization concentrates on the join operators. The projections can be added as needed, i.e., whenever an operator materializes its input (and thus breaks the processing pipeline), all attributes that are no longer needed are projected away. Selections are added greedily, i.e., a selection is applied as early as possible. The rationale is that most selection predicates are cheap to evaluate and the selections reduce the work required by the following operators. Thus the optimizer only has to order the join/cross-product operators, and the other operators are added greedily. Unfortunately the problem remains NP hard even with this simplification.

The problem of finding the optimal join order can be seen as finding the optimal binary tree whose leaves correspond to the relations in the *from* clause. The inner nodes are joins or cross products, depending on available predicates suitable for a join, and are determined implicitly by the relations involved in their subtrees. Therefore, only the structure of the binary tree and the labeling of the leaf nodes has to be specified by the query optimizer. However the number of binary trees with n leaf nodes is $\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ are the Catalan Numbers, which grow in the order of $\Theta(4^n/n^{\frac{3}{2}})$. As this grows very fast, some approaches reduce the search space by considering only a limited set of binary trees. A popular restriction is the limitation to *left-deep* join trees (or more general to *linear* join trees). A linear join tree is a join tree where only one of the two subtrees of a join operator may contain other join operators. If only the left subtree may contain other join operators, the trees are called left-deep. Figure 2 shows an example. General join trees without restrictions are called *bushy* join trees. Left-deep join trees are attractive (e.g., System R used them), as they are potentially easier to execute and the number of left-deep join trees is much smaller than the number of bushy join trees. As there are $n!$ ways to label the leaf nodes of the join tree for n relations, there are $n!\mathcal{C}(n-1)$ bushy trees, but "only" $n!$ left-deep trees. Unfortunately the optimal join tree can be a bushy tree, and these cases are not uncommon, which means that generating only left-deep trees can hurt query execution performance. Most modern query optimizers therefore construct bushy join trees.

The huge factor of $n!$ is caused by the fact that all combinations of relations are considered valid. The search space can be significantly reduced by avoiding the creation of cross products. When combining two relations, the optimizer can either use a cross-product, or use a join if there is a suitable join predicate in the *where* condition. Joins are much more efficient than cross-products, and although it is sometimes beneficial to use cross-products between separate relations, these cases are rare. When allowing cross-products, any relations can be combined, otherwise combinations are only possible if a suitable join predicate exists. The join possibilities implied by the query are captured in the query graph, as shown in Fig. 3. The relations from the *from* clause form the nodes, while the potential join conditions form the edges. Now the optimizer only has to consider sub-problems, i.e., sets of relations, that are connected in the query graph (this assumes that the



Query Optimization (in Relational Databases). Figure 2. Left-deep Versus bushy join trees.

query graph itself is connected, which can be guaranteed by adding additional edges). For example the relations A,B,C can be joined and thus could be part of a join tree, while no join tree will consist only of B,C,D, as this would require a cross-product. The structure of the query graph greatly affects the size of the search space. If the query graph forms a chain, the join ordering problem is no longer NP hard and can be solved in $O(n^3)$. If the query graph forms a clique, the problem is still NP hard (and just as difficult as when including cross products). Most queries are between those two extremes, and more like a chain than like a clique, and can thus be optimized much more efficiently by avoiding cross products.

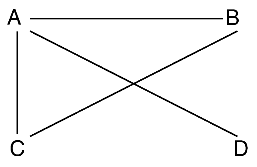
Putting these observations together, SPJ queries can be optimized by the following strategy:

1. The only optimization decision is the join order, selections and projections are added greedily
2. The relations in a join tree must be connected in the query graph
3. When constructing left-deep trees, the right hand side of a join must contain only one relation.

The seminal System R paper on query optimization [9] introduced a dynamic programming (DP) strategy to optimize the join order. A slightly generalized version that generates bushy trees is shown in Fig. 4. It computes the optimal join order of the relations $R_1,...,R_n$ (ignoring selections and projections for a moment). The basic strategy is to construct solutions for larger (sub-)problems (i.e., problems involving more relations) from optimal solutions of smaller problems. For example, consider the top-most join in a join tree with four relations. It will either combine two join trees with two relations each or a single relation with a join tree with three relations. The problem of joining four relations can thus be expressed as combining smaller problems with one to three relations each. Accordingly, the DP table is first organized by the size (i.e., number of relations) of a problem. For a given size, the table stores the optimal execution plan for a given set of relations. In lines 1–2 the algorithm initializes the DP table by adding table scans as the optimal solution for problems involving a single relation. The loop starting in line 3 now creates solutions of size s by combining smaller solutions. Lines 4–5 find all pairs of problems already solved (S_1, S_2) that have a combined size of s. If S_1 and S_2 overlap (i.e., they have relations in common) the pair is ignored, as no valid join tree can be constructed (line 6). Similarly (S_1, S_2) is ignored if S_1 and S_2 are not connected in the query graph to avoid the creation of cross-products (line 7). Otherwise it is possible to construct a new execution plan p that joins the known solutions for S_1 and S_2 (line 8). If no solution for $S_1 \cup S_2$ is known, (or the estimated costs of the new plan are less than for the currently known solution), p is added as a solution of size s for $S_1 \cup S_2$ in the DP table (lines 9–10). At the end of the algorithm the DP table entry for size n contains the optimal solution (line 11).

```
Select   *
from     Aa, Bb, Cc, Dd
where    a.a1 = b.b1 and
         a.a2 = c.c1 and
         a.a3 = d.d1 and
         b.b2 = c.c2
```

SQL query



Query Optimization (in Relational Databases). Figure 3. A query and its query graph.

```
DPsize($R = \{R_1,...,R_n\}$)
for each $R_i \in R$
    dpTable[1][{$R_i$}] = $R_i$
for each $1 < s \leq n$ ascending // size of plan
    for each $1 \leq s_1 < s$ // size of left subplan
        for each $S_1 \in$dpTable[$s_1$], $S_2 \in$dpTable[$s - s_1$]
            if $S_1 \cap S_2 \neq \emptyset$ continue
            if $\neg$($S_1$ connected to $S_2$) continue
            $p$ = dpTable[$s_1$][$S_1$]$\bowtie$dpTable[$s - s_1$][$S_2$]
            if dpTable[$s$][$S_1 \cup S_2$]=$\emptyset$ V cost($p$)<cost(dpTable[$s$][$S_1 \cup S_2$])
                dpTable[$s$][$S_1 \cup S_2$] = p
return dpTable[$n$][{$R_1,...,R_n$}]
```

Query Optimization (in Relational Databases). Figure 4. Dynamic programming strategy for SPJ queries.

The algorithm in Fig. 4 is simplified, as it ignores selections and projections. They can be added greedily in lines 2 and 8 and do not affect the algorithm otherwise. A more complex part that is missing is the selection of the physical join operator. Line 8 simply states $\bowtie$, but in a real system, there are multiple join operators available, typically at least nested-loop join, hash join, and sort-merge join. Lines 8–10 should therefore loop over the different join implementations and try all of them. What complicates this choice is that the different implementations behave differently, in particular the sort-merge join. When the input has to be sorted, the sort-merge join is relatively expensive, but when it is already sorted it is very cheap. And the output of a sort-merge join is itself sorted, which can render a following sort-merge join cheap if the order can be reused. Tuple orders that could be used by other operators are called *interesting orderings*, and the set of interesting orderings for the current query is computed before starting the *plan generation* (i.e., the DP algorithm, which constructs execution plans). During plan generation, plans that are more expensive than others have to be preserved, but provide an interesting order the others do not. This can be generalized by the concept of *physical properties*. A physical property is a characteristic of a plan that affects its runtime behavior (i.e., the costs of subsequent operators), but not its logical equivalence. The physical properties define a partial order between plans, describing which plans satisfy "more" properties. For example a plan with sorted output dominates another plan with unsorted output (concerning the physical property "ordered"), while two plans with differently sorted output are not directly comparable. During plan generation, a plan only dominates another plan if both the physical properties are dominating and the estimated costs are lower. As a consequence, the DP table entries no longer consist of single optimal plans, but of sets of plans, in which no plan dominates the other. Note that physical properties are an example for re-establishing the principle of optimality required for dynamic programming, which is also required in other contexts.

More Complex Queries

While Selection-Projection-Join queries are a important class of queries, queries can be much more complex. When only optimizing (inner) joins, the joins are freely reorderable, which means that any join order is valid as long as syntax constraints are satisfied (i.e., the relations required for the join predicates are available). This is no longer the case for other operators like outer joins and aggregations. A simple approach to handle these is to split the query into blocks that are freely reorderable (e.g., above and below an outer join) and to optimize the blocks individually. But this is too restrictive, and outer joins still allow for some reorderings, which have been described as algebraic equivalences. The challenge is to incorporate these equivalences into a cost-based (and potentially constructive) query optimizer. For outer joins, it has been shown how the possible reorderings can be expressed as dependencies on input relations [8]. The algorithm analyzes the original query and computes the set of relations that have to be part of a join tree before a specific outer join is applicable. Using this information the outer joins can be integrated easily, the plan generator just has to check the additional requirements before inserting an outer join. The main difficulty is computing this dependency information, but once it is available, the optimization is relatively simple. Other operators like aggregations are more difficult to integrate. Aggregations can be moved down a join if the join itself does not affect the aggregation results (e.g., a 1:1 join involving the grouping attribute where the join behaves like a selection). Pushing the aggregation down can reduce the effort for the join itself and might thus be beneficial. But in cases where the aggregation can simply be moved are relatively limited. Here, a more general movement of aggregations is possible by allowing for compensation actions. These are computations added to the plan that compensate the fact that the aggregation is performed at a different position. Adding these kinds of optimizations into a cost-based, constructive query optimizer is very challenging, which is why they are usually implemented as heuristical rewrite operations.

Another important aspect of optimizing complex queries is unnesting nested queries and the related problem of view resolution. The SQL query language allows for nesting queries inside other queries, either explicitly by including a nested *select* block, or implicitly by accessing a view. The nested query could be optimized independently from the outer query, and then during the optimization of the outer query treated like a base relation with specific costs. But this will often lead to inferior plans, for example if selection predicates from the outer query could be pushed down into a view. Instead, the optimizer tries to merge the nested query with the outer query into one flat query, which is then optimized in one step. Perhaps even more important than the unified optimization is a decoupling between the nested query and the outer query. The evaluation of the nested query can depend on the attributes of the outer query, which implies a very expensive nested-loop evaluation. In many cases it is possible to unnest these queries such that the nested (and apparently dependent) part can be evaluated independently, and then joined appropriately with the outer part of the query [9]. These optimizations greatly improve the evaluation of nested queries.

Key Applications

Query optimization techniques can improve the query execution time by orders of magnitude. All modern relational database systems therefore implement at least some optimization techniques.

Cross-references

- ▶ Parallel data placement
- ▶ Parallel database management
- ▶ Parallel query execution algorithms
- ▶ Parallel query processing

Recommended Reading

1. Chaudhuri S. An overview of query optimization in relational systems. In Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1998, pp. 34–43.
2. Garcia-Molina H., Ullman J.D., and Widom J. Database system implementation. Prentice-Hall, 2000.
3. Graefe G. The cascades framework for query optimization. Q. Bull. IEEE TC on Data Engineering, 18(3):19–29, 1995.
4. Graefe G. and McKenna W.J. The volcano optimizer generator: Extensibility and efficient search. In Proc. 9th Int. Conf. on Data Engineering, 1993, pp. 209–218.
5. Haas L.M., Freytag J.C., Lohman G.M., and Pirahesh H. Extensible query processing in starburst. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1989, pp. 377–388.
6. Moerkotte G. Building query compilers, available at http://db.informatik.uni-mannheim.de/moerkotte.html.en, 2006.
7. Muralikrishna M. Improved unnesting algorithms for join aggregate SQL Queries. In Proc. 18th Int. Conf. on Very Large Data Bases, 1992, pp. 91–102.
8. Rao J., Lindsay B.G., Lohman G.M., Pirahesh H., and Simmen D.E. Using EELS, a practical approach to outerjoin and antijoin reordering. In Proc. 17th Int. Conf. on Data Engineering, 2001, pp. 585–594.
9. Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., and Price T.G. Access path selection in a relational database management System. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 23–34.

Query Optimization for Multidimensional Systems

▶ Query Processing in Data Warehouses

Query Optimization in Distributed Database Systems

▶ Distributed Query Optimization

Query Optimization in Sensor Networks

KIAN-LEE TAN
National University of Singapore, Singapore, Singapore

Synonyms

In-network query processing

Definition

Query optimization is the process of producing a query evaluation plan (QEP) for a query that minimizes or maximizes certain objective functions. A query in a sensor network has additional clauses that specify the life time of a query, the frequency in which the sensor data should be monitored, and even the rate in which query answers should be returned. As such, the query plan must reflect these. In addition, a typical query plan comprises two main components: a communication component that sets up the communication structure for data delivery, and a computation component that performs the operation in the sensor network and/or the root node. Because sensor nodes are low-powered, besides minimizing computation cost, optimization criterion include minimizing energy consumption (e.g., by minimizing transmission cost) or maximizing the lifespan of the entire sensor network. As such, the cost model must consider these various factors.

Historical Background

Query optimization has always been an important area of research in database query processing. This is because a poorly chosen query execution plan can result in significant waste of resources, and more importantly, user dissatisfaction. In sensor network, the problem is more critical because of the resource constrains of the sensor nodes which are limited in computation power, bandwidth, memory, and energy.

Most of the existing works focused on in-network aggregation [2,3,5,6,10]. While work in [3,10] aim at precise answers, work in [2,6] assume errors (approximations) can be tolerated. In particular, the work in [6] exploits the tradeoff between data quality and energy consumption to extend the lifetime of the sensor network. Both the Cougar project [10] and the TinyDB project [3], in their prototype design of a sensor network management system, offer broader insights into query processing and optimization issues.

Foundations

To support applications in sensor networks, the database community has viewed the sensor network as a database [3,10]. This provides a good logical abstraction for sensor data management. Users can issue declarative queries without having to worry about how the data are generated, processed, and transferred within the network, and how sensor nodes are (re)programmed. As such, query optimization techniques can also be applied to optimize the network operations.

Queries

Queries are typically expressed using an extended SQL that include additional clauses that specify the duration and sampling rates. As an example, the following query counts the number occupied nests in each loud region of a certain island [3,10]:

```
SELECT region, CNT(occupied), AVG
(sound)
FROM sensors
GROUP BY region
HAVING AVG(sound) > 200
SAMPLE PERIOD 10 FOR 3,600
```

Here, the clause "SAMPLE PERIOD ... FOR" indicates that the sensors will sample the environment every 10 seconds for a duration of 3,600 seconds.

Query Plan

To evaluate a query, a query plan has to be generated for the sensors. A query plan specifies the role of each sensor (the computation to be performed, the rate at which it should sample the data), and the communication structure between sensors. For the sample query above, three alternative plans can be considered: (i) Each sensor samples its data every 10 seconds, and then transmits the data back to the base station; at the base station, it will perform the grouping and aggregate computation as in a centralized system. (ii) A sensor within a region is selected as a cluster head (CH); all sensors within the region send their sampled data to the corresponding CH; the CH performs the aggregate and sends it back to the base station if it satisfies the HAVING clause (i.e., >200). (iii) This is similar to Plan 2, except that sensors within a region can be further hierarchically partitioned so that each partition has a leader that performs partial aggregation of the sampled data from its child nodes.

For Plan 1, each sensor node takes on at most two roles: (i) sample the data and transmit the sampled value back to the base station, and (ii) relay the data it receives from its child nodes if it is an internal node along the path to the base station. For Plan 2, the CH has the additional computation task of performing the aggregate and determining whether it should be routed back to the base station. For Plan 3, leaders have the responsibility to perform partial aggregates.

All the plans may also specify the sensor nodes in which one should be transmitting data to and/or receiving data from, i.e., the communication structure.

Clearly, there are many other possible plans that can be generated, and it is the optimizer's task to pick the one that best suits the objectives.

Metadata

The optimizer makes its decision based on certain metadata. For example, the metadata for each sensor include the static information about its location (region), the sensor types, the amount of memory, the energy consumption per operation type, the energy consumption per sampling per sensor type, and so on. It may also be necessary to maintain dynamic statistics on a sensor's estimated (remaining) battery life, and the selectivity of query predicates. These metadata are periodically collected (via the routing tree) by the optimizer, and used in several ways. For example, for sensor nodes with short battery life, the sampling rate may be adjusted so that they can operate in a doze mode longer in order to extend the lifespan of the entire network. As another example, nodes with long battery life may be selected as the cluster heads or leaders, nodes with moderate amount of energy can relay messages, and nodes with low battery power will simply transmit their sampled data. As such, it is possible to balance the energy across all nodes. As another example, knowing the energy consumption per sampling per sensor type may allow the optimizer to reorder predicates on different sensor types.

Cost Model

For each query plan, the optimizer estimates the cost of the plan. The cost model depends on the objective function. In a sensor network environment, there are two key metrics: energy consumption and transmission cost (as communication consumes most energy).

As an example, consider an arbitrary sensor node along the path of the routing tree. Let N be the number of sensor types and k be the number of predicates. Let E_i is the cost to sample sensor type i, E_{trans} and E_{recv} be the energy to transmit and receive a message respectively, E^i_{pred} be the energy consumed to evaluate predicate i, and E_{agg} be the energy to compute a (partial) aggregate, and C be the number of child nodes routing through this node. The energy at a node s to collect a sample, and transmit its partial aggregate, including the costs to forward data, can be estimated as follows:

$$e_s = \sum_{i=0}^{N} E_i + \sum_{i=0}^{N} E^i_{pred} + E_{recv} \times C + E_{agg} + E_{trans}$$

The energy consumed by the node is the cost to read all the sensors at the node (first component), plus the cost to evaluate the predicates (second component), plus the cost to receive partial aggregates from its child nodes (third component), plus the cost to compute a partial aggregate (fourth component), plus the cost to transmit the partial aggregate (fifth component).

Suppose sensor s has a remaining battery capacity of B_s Joules. Then, s has enough power to last B_s/e_s sample collections. To extend the lifespan of the system, a plan that maximizes this value has to be determined. Let there be P plans, then a plan that maximizes the minimim number of sample collections is the one that should be selected (to maximize the lifespan of the system), i.e.,

$$max^P_{i=1}(min^N_{j=1} B_j/e_j)$$

Depending on different objectives and model, similar cost models can be derived.

Centralized Optimization

Most of the existing work on optimizing sensor network queries are based on a centralized optimizer, i.e., the optimizer at the base station determines the query plan [3,10]. There are two dimensions in which query processing can be optimized. The first dimension deals with the grouping of the sensor nodes to support in-network computation (e.g., aggregates). Essentially, the optimizer enumerates the possible alternative plans based on different ways in which nodes are grouped. Some heuristics are:

1. On one extreme, all nodes belong to a single group;
2. On another extreme, each node forms a single group;
3. Between the two extremes, different group-based heuristics can be adopted: proximity-based schemes cluster nodes that are close-by together, semantic-based schemes cluster nodes that are semantically related, e.g., same set of sensor types, same resources, same metadata, etc.

The second dimension covers heuristics that are used to reorder the operations of a query. Some of these are:

1. When a sensor node is required to sample multiple attributes (temperature, humidity, light) and the query involves a number of predicates, different orderings of the samping and predicate evaluations may result in different energy consumption. This is the case because sampling consumes more power than evaluating a predicate. As such, to conserve energy, it makes sense to order selective predicates first before the non-selective predicates; in addition, data should be sampled only when necessary. Intuitively, once a sampled value is discarded (because it does not satisfy a predicate), there is no need to expend energy to sample other attributes. Thus, the optimizer enumerates different sequences of sampling and predicate evaluations to find the one that is most energy efficient. As an example, consider a query that finds sensor nodes whose accerlerometer and magnetometer readings exceed thresholds a1 and m1 respectively. As the magnetometer consumes 50 times more energy than the accerlerometer to sample a reading, if the selectivity of the predicate on the accerlerometer reading is low, then it makes sense to sample the accerlerometer first. Moreover, the magnetometer should only be sampled when the accerlerometer reading is larger than a1.
2. For certain event monitoring queries, they can be rewritten into a window join queries that can be more efficiently processed in the network.

Distributed Optimization

As centralized optimization schemes require certain metadata that must be periodically obtained from the sensor nodes, it may be costly (in terms of power) to collect this information. An alternative approach is to perform distributed optimization [4]. Here, the basic idea is to identify clusters of nodes with its associated cluster head (CH) so that these CHs optimize the queries for processing within the cluster. Thus, the base stations disseminate the query to the CHs, each of which optimizes the query based on the local metadata and controls the processing within the cluster.

With distributed optimization, the metadata is collected at the CH, rather than the base station. This means that the transmission overhead for metadata is reduced. Moreover, "local" metadata are more accurate as some globally collected metadata may be too coarse (e.g., distribution of data). However, the CHs incur the overhead of optimizing the query.

Here, the challenge is to determine the clusters (either spatially or semantically). In spatial-based clustering scheme, the number of groups are based on the radius of a group, which is the number of hops between a non-CH node and its CH. In the semantic-based clustering scheme, nodes with similar netadata are formed into groups.

Key Applications

Sensor databases can be applied in many applications, e.g., environmental monitoring, military surveillance. To process queries in these applications, it is necessary to optimize the queries in order to ensure that resources are well utilized, and the system lifespan can be sufficiently large to minimize any need to replace the batteries regularly.

Future Directions

Most of the existing work in the literature focused on optimizing aggregate queries one at a time. Recently, join queries have been studied. In [1], static join queries are considered, and in [9], continuous join queries are examined. In both cases, join processing is pushed into the sensor network. In the former, sensor nodes are grouped so that a static table can be partitioned across them; new sampled records are broadcast within a group, and only answers need to be sent back to the base stations. In the latter, the focus is on minimizing the number of subqueries that need to be injected into the sensor network. As sensor nodes become more powerful, there is much opportunity for optimizing complex queries that have not been previously studied. Another promising direction that has recently received attention is the multi-query optimization problem [7,8]. Here, multiple queries submitted by users are optimized collectively to exploit commonality among them so that any redundant data accesses from the sensors can be eliminated. Current focus is on non-join queries. Extending these schemes to handle more complex queries is an interesting direction to explore. Yet another direction is to consider a large scale sensor network deployment that involves multiple base stations. Here, the challenge is to optimize the network lifespan as well as the load across the base stations.

Cross-references

▶ Continuous Queries in Sensor Networks
▶ Data Aggregation in Sensor Networks
▶ In-Network Query Processing
▶ Sensor Networks

Recommended Reading

1. Abadi D.J., Madden S., and Lindner W. REED: robust, efficient filtering and event detection in sensor networks. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 769–780.
2. Deligiannakis A., Kotidis Y., and Roussopoulos N. Hierarchical in-network data aggregation with quality guarantees. In Proc. 9th Int. Conf. on Extending Database Technology, 2004, pp. 658–675.
3. Madden S., Franklin M.J., Hellerstein J.M., and Hong W. TINYDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst., 30(1):122–173, 2005.
4. Rosemark R. and Lee W.-C. Decentralizing query processing in sensor networks. In Proc. 1st Annual Int. Conf. on Mobile and Ubiquitous Syst., 2005, pp. 270–280.
5. Silberstein A. and Yang J. Many-to-many aggregation for sensor networks. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 986–995.
6. Tang X. and Xu J. Extending network lifetime for precision constrained data aggregation in wireless sensor networks. In Proc. 25th Annual Joint Conf. of the IEEE Computer and Communications Societies, 2006.
7. Trigoni N., Yao Y., Demers A.J., Gehrke J., and Rajaraman R. Multi-query optimization for sensor networks. In Proc. 1st IEEE Int. Conf. on Dist. Comput. in Sensor Syst., 2005, pp. 307–321.
8. Xiang S., Lim H.B., Tan K.L., and Zhou Y. Two-tier multiple query optimization for sensor networks. In Proc. 23rd Int. Conf. on Distributed Computing Systems, 2007, p. 39.
9. Yang X., Lim H.B., Ozsu T., and Tan K.L. In-network execution of monitoring queries in sensor networks. In Proc. ACM SIGMOD Int. Conf. on Management of Data., 2007, pp. 521–532.
10. Yao Y. and Gehrke J. Query processing in sensor networks. In Proc. 1st Biennial Conf. on Innovative Data Systems Research, 2003.

Query Parallelism

► Intra-query parallelism

Query Plan

Evaggelia Pitoura
University of Ioannina, Ioannina, Greece

Synonyms

Query plan; Query execution plan; Query evaluation plan; Query tree; Operator tree

Definition

A query or execution plan for a query specifies precisely how the query is to be executed. Most often, the plan for a query is represented as a tree whose internal nodes are operators and its leaves correspond to the input relations of the query. The edges of the tree indicate the data flow among the operators. The query plan is executed by the query execution engine.

Key Points

During query processing, an input query is transformed into an internal representation, most often, into a relational algebra expression. To specify how to evaluate a query precisely, each relational operator is then mapped to one or more physical operators, which provide several alternative implementations of relational operators.

A query plan for each query specifies the physical operators to be used for its execution and the order of their invocation. Most commonly, a query plan is represented as a tree. The leaf nodes of the query tree correspond to the input (or base) database relations of the query and its internal nodes to physical operators. The edges of the tree specify the data flow among the operators. Each operator receives input from its child nodes in the tree and, in turn, its output is used as input to its parent node.

The query optimizer enumerates alternative query plans for each query and selects the most efficient among them using a cost estimation model. The selected query plan is then passed for execution to the query execution engine that results in generating answers to the query.

Query plans can be divided into prototypical shapes and query execution engines can be divided into groups based on which shapes of plans they can evaluate. Such shapes include left-deep, right-deep and bushy plans. Deep plans are plans in which each join involves at least one base relation, whereas bushy plans are more general in that a join could involve one or two base relations or the results of one or two other join operations. Finally, for queries with common subexpressions, the query evaluation plan may be an acyclic directed graph (DAG) instead of a tree.

Cross-references

► Evaluation of Relational Operators
► Query Optimization
► Query Processing

Recommended Reading

1. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73–170, 1993.
2. Ramakrishnan R. and Gehrke J. Database Management Systems. McGraw-Hill, New York, 2003.

Query Planning and Execution

► Query Processing and Optimization in Object Relational Databases

Query Point Movement Techniques for Content-Based Image Retrieval

Kien A. Hua, Danzhou Liu
University of Central Florida, Orlando, FL, USA

Definition

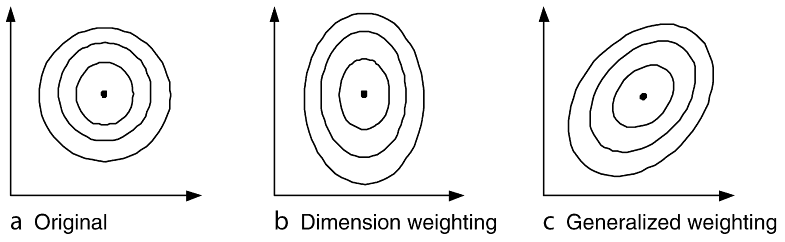
Target search in content-based image retrieval (CBIR) systems refers to finding a specific (target) image such as a particular registered logo or a specific historical photograph. To search for such an image, query point movement techniques iteratively move the query point closer to the target image for each round of the user's relevance feedback until the target image is found. The goals of query point movement techniques include avoiding local maximum traps, achieving fast convergence, reducing computation overhead, and guaranteeing to find the target.

Historical Background

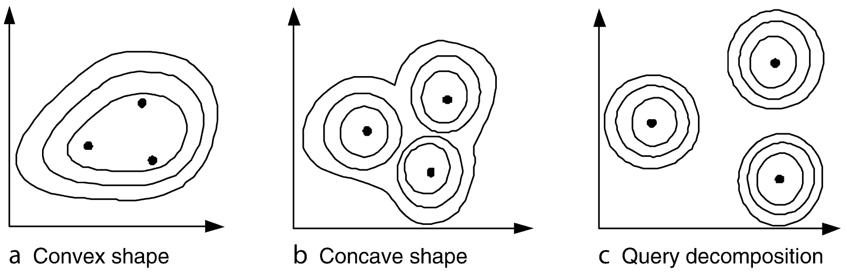
Images in a database are characterized by their visual features, and represented as points in a multidimensional feature space. A *query point* is one of these image points, selected to find similar images represented by image points nearest to the query point in the feature space. This cluster of nearby or relevant image points has a shape (see Figs. 1 and 2) referred to as the *query shape*.

For each iteration, a query point movement technique attempts to move the query point closer to the target image by refining the query based on user's relevance feedback. Existing query point movement techniques can be divided into two categories: single-point and multiple-point movement techniques. A technique is classified as a single-point movement technique if the refined query Q_r at each iteration consists of only one query point; otherwise, it is a multiple-point movement technique. In the latter category, the query result is the set of images nearest to the set of query points. Typical query shapes of single-point movement and multiple-point movement techniques are illustrated in Figs. 1 and 2, respectively, where the contours represent equi-similarity surfaces. Single-point movement techniques, such as MARS [9] and MindReader [5], construct a single query point close to relevant images and away from irrelevant ones. MARS uses a weighted distance (producing shapes as shown in Fig. 1b), where each dimension weight is inversely proportional to the standard deviation of the relevant images' feature values in that dimension. The rationale is that a small variation among the values is more likely to express restrictions on the feature, and thereby should carry a higher weight. On the other hand, a large variation indicates that this dimension is not significant in the query, and should thus assume a lower weight. MindReader achieves better results by using a generalized weighted distance, see Fig. 1c for its shape.

In multiple-point movement techniques such as Query Expansion [1], Qcluster [6], and Query Decomposition [4], multiple query points are used to define the ideal space that is most likely to contain relevant results. Query Expansion groups query points into clusters and choose their centroids as the representatives of the query Q_r (see Fig. 2a). The distance of a point to Q_r is defined as a weighted sum of individual distances to those representatives. The weights are proportional to the number of relevant objects in the clusters. Thus, Query Expansion treats local clusters differently, compared to the equal treatment in single-point movement techniques. In some queries, clusters



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 1. Single-point movement query shapes.



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 2. Multiple-point movement query shapes.

are too far apart for a unified, all-encompassing contour to be effective; separate contours can yield more selective retrieval. This observation motivated Qcluster to employ an adaptive classification and cluster-merging method to determine optimal contour shapes for complex queries. Qcluster supports disjunctive queries, where similarity to any of the query points is considered relevant (see Fig. 2b). To bridge the semantic gap more effectively, A Query Decomposition technique was presented in [4]. Based on user's relevance feedback, this scheme automatically decomposes a given query into localized subqueries, which more accurately capture images with similar semantics but with very different appearance (see Fig. 2c).

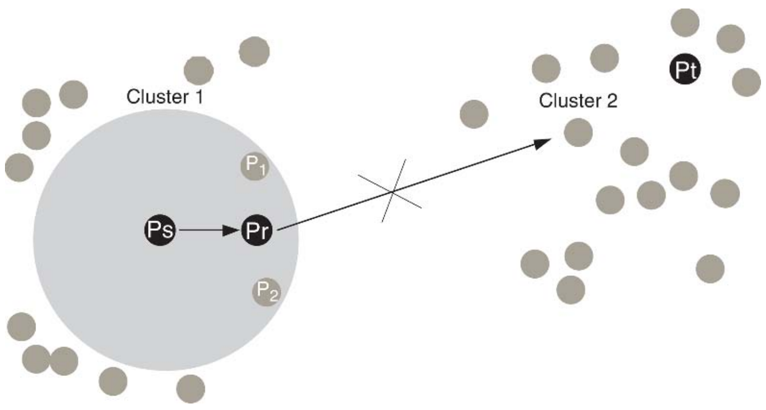
Standard query point movement techniques, explained above, allow re-retrieval of previously determined relevant images when they fall in the search range again. This leads to two major disadvantages:

1. *Local maximum traps*. Since query points in relevance feedback systems have to move through many regions before reaching a target, it is possible that they get trapped in one of these regions. Figure 3 illustrates a possible scenario where p_s and p_t denote the starting query point and the target point, respectively. As a result of a 3-NN search at p_s, the system returns points p_1 and p_2, in addition to query point p_s. Since both p_1 and p_2 are relevant, the refined query point p_r is their centroid and the anchor of the next 3-NN search. In this situation, the system will retrieve exactly the same set; from which, points p_1 and p_2 are again selected. In other words, the system can never get out of the subspace because the retrieval set is saturated with the k selected images. Although the system can escape with a larger k, it is difficult to guess a proper threshold. Consequently, the user might not even know a local maximum trap is occurring, and there is no guarantee to find the target image.
2. *Slow convergence*. The centroid of the relevant points is typically selected as the anchor of refined queries. This, coupling with possible retrieval of already visited images, prevents aggressive movement of the search process (see Fig. 4, where $k = 3$). Slow convergence incurs longer search time, and significant computation and disk access overhead.

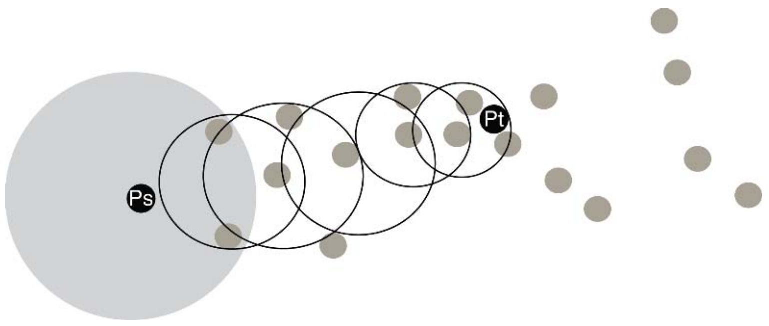
Foundations

To address the limitations of standard query point movement techniques, four target search methods have been proposed [8]: Naïve Random Scan (NRS), Local Neighboring Movement (LNM), Neighboring Divide and Conquer (NDC), and Global Divide and Conquer (GDC) methods. All these methods are designed around a common strategy: they do not retrieve previously selected images (i.e., shrink the search space). Furthermore, NDC and GDC exploit Voronoi diagrams to aggressively prune the search space and move towards the target image faster.

More formally, a query in target search is defined as $Q = \langle n_Q, P_Q, W_Q, D_Q, \mathbb{S}, k \rangle$, where n_Q denotes the number of query points in Q, P_Q the set of n_Q query points in the search space $\mathbb{S}$, W_Q the set of weights associated with P_Q, D_Q the distance function, and k the number of points to be retrieved in each iteration. Using these notations, the four target search techniques can be described as follows:



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 3. Local maximum trap.



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 4. Slow convergence.

Naïve Random Scan Method (NRS)

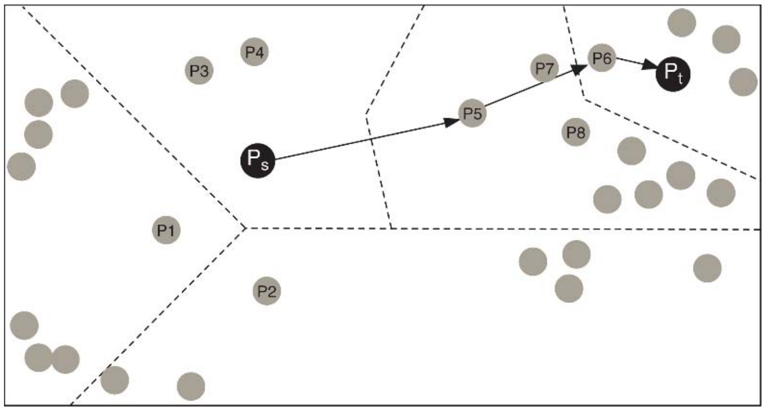
This method randomly retrieves k different images at a time until the user finds the target image or the remaining set is exhausted. Specifically, at each iteration, a set of k not previously selected images are randomly retrieved from the candidate set $\mathbb{S}'$ for relevance feedback, and $\mathbb{S}'$ is then reduced by k for the next iteration. Clearly, this strategy does not suffer local maximum traps and is able to locate the target image after some finite number of iterations. In the best case, NRS takes one iteration, while the worst case requires $\left\lceil \frac{|\mathbb{S}|}{k} \right\rceil$. On average, NRS can find the target in $\left\lceil \sum_{i=1}^{\lceil \frac{|\mathbb{S}|}{k} \rceil} i / \left\lceil \frac{|\mathbb{S}|}{k} \right\rceil \right\rceil = \left\lceil \left(\left\lceil \frac{|\mathbb{S}|}{k} \right\rceil + 1 \right)/2 \right\rceil$ iterations. In other words, NRS takes $\mathcal{O}(|\mathbb{S}|)$ to reach the target point. Therefore, NRS is only suitable for a small database set.

Local Neighboring Movement Method (LNM)

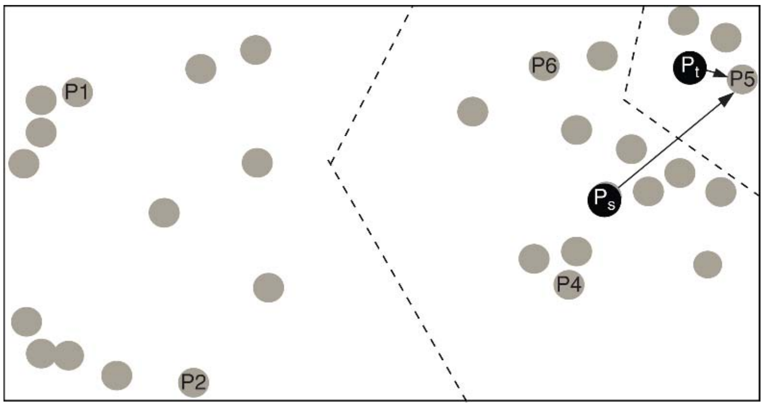
This method applies the non-re-retrieval strategy to MindReader [5]. Specifically, Q_r is constructed so that it moves towards neighboring relevant points and away from irrelevant ones, and k-NN query is now evaluated against $\mathbb{S}'$ instead of $\mathbb{S}$. When LNM detects a local maximum trap, it requests that the user selects the most relevant image. This way, LNM can overcome local maximum traps, although it could take many iterations to do so. Again, one iteration is required in the best case. If data is uniformly distributed in the n-dimensional hypercube, the worst and average cases are $\left\lceil \sqrt{n}\sqrt[n]{|\mathbb{S}|} / \lceil \log_{2^n} k \rceil \right\rceil$ and $\left\lceil \left(\frac{\sqrt{n}\sqrt[n]{|\mathbb{S}|}}{\lceil \log_{2^n} k \rceil} + 1 \right)/2 \right\rceil$, respectively. If the data are arbitrarily distributed, then the worst case could be as high as that of NRS, i.e., $\left\lceil \frac{|\mathbb{S}|}{k} \right\rceil$ iterations (e.g., when all points are on a line). In summary, in the worst case LNM could take anywhere from $\mathcal{O}(\sqrt[n]{|\mathbb{S}|})$ to $\mathcal{O}(|\mathbb{S}|)$.

Neighboring Divide and Conquer Method (NDC)

Although LNM can overcome local maximum traps, it does so inefficiently, taking many iterations and in the process returning numerous false hits. To speed up convergence, Voronoi diagrams are used in NDC to reduce the search space after each round of relevance feedback. That is, the k-NN search in each iteration is performed only within the Voronoi cell containing the query point. It can be proved that the target image must reside in this Voronoi cell [8]. Since this strategy aggressively prunes the search space and moves rapidly towards the target image, it can overcome local maximum traps and achieve fast convergence. Figure 5 illustrates how NDC approaches the target after pruning the search space three times. In the first iteration, points p_1, p_2, and p_s are randomly chosen by the system, assuming $k = 3$; and they are used to construct a Voronoi diagram partitioning the search space into three regions. The user identifies p_s as the most relevant point (i.e., most similar to the target image). Since the target image must reside in the Voronoi cell containing p_s, the computation of the k-NN query anchored at p_s can be confined to this cell while the other two Voronoi regions can be safely ignored. This step retrieves three new nearest neighbors p_3, p_4, and p_5. Their Voronoi diagram further partitions the current search space into three regions. The user again correctly selects p_5 as the most relevant point, and therefore the query point for the third iteration. This refined query results in another set of relevant points p_6, p_7, and p_8; and another Vonronoi diagram is constructed. This time, the user selects p_6 as the most relevant image. Using it



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 5. Example of NDC.

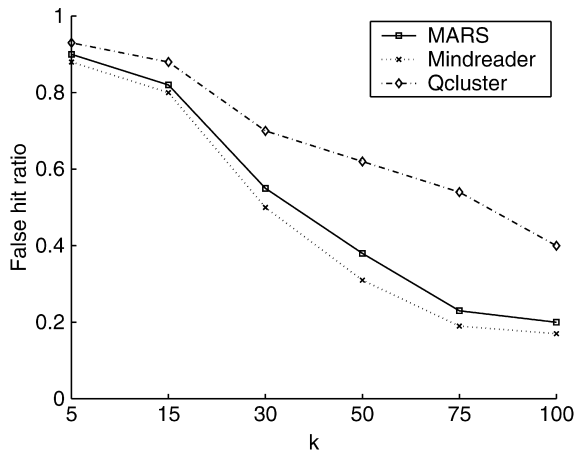


Query Point Movement Techniques for Content-Based Image Retrieval. Figure 6. Example of GDC.

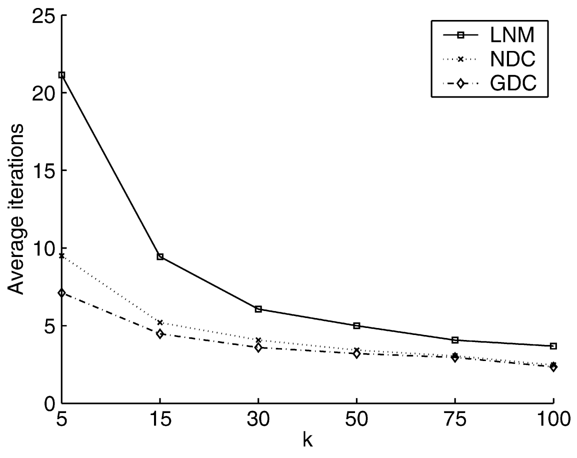
as the query point for the fourth iteration, the system returns three relevant points and the user identifies p_t as the target image. If the data points are uniformly distributed, NDC reaches the target point in no more than $\mathcal{O}(\log_k|\mathbb{S}|)$ iterations. When $\mathbb{S}$ is arbitrarily distributed, the worst case could take up to $\lceil\frac{\mathbb{S}}{k}\rceil$ iterations (e.g., all points are on a line), the same as that of NRS. In other words, NDC could still require $\mathcal{O}(|\mathbb{S}|)$ iterations to reach the target point in the worst case.

Global Divide and Conquer Method (GDC)

To reduce the number of iterations in NDC under the worst case scenario, GDC constructs the Voronoi diagram based on points randomly selected from the current search space, instead of using points from the query result. An example is given in Fig. 6. In the first iteration, a Voronoi diagram is constructed based on three randomly sampled points p_1, p_2, and p_s, assuming $k = 3$. The user selects p_3 as the most relevant point, and this results in p_4, p_5, and p_6 as the query result as computed in NDC. The user now selects p_5 as most relevant in the second iteration. The system randomly selects three points in the Voronoi cell associated with p_5, and uses them to further partition this cell into three smaller Voronoi cells. The computation of the k-NN query anchored at p_5 over the smaller Voronoi cell containing p_5 returns three new points, and the user identifies p_t as the target image in the third round. As proved in [8], the worst case for GDC is bounded by $\mathcal{O}(\log_k|\mathbb{S}|)$. This implies that for arbitrarily distributed datasets, GDC converges faster than NDC in general, although NDC might be as fast as GDC for certain queries, e.g., the starting query point is close to the target point. In the previous example (Fig. 5), NDC could also take three iterations, instead of four, to reach the target point if the initial k points were the same as in Fig. 6, as opposed to Fig. 5.



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 7. False hit ratio.



Query Point Movement Techniques for Content-Based Image Retrieval. Figure 8. Average iterations.

Key Applications

Multimedia Search Engine, Crime Prevention, Graphic Design.

Future Directions

1. Incorporate information from the log file on user relevance feedback to determine the query results in each feedback iteration, instead of performing the traditional *k*-NN computation. This strategy can minimize the effect of the semantic gap between the low-level visual features and the high-level concepts in the images.
2. Multiple query points, as in standard query point movement techniques such as Query Expansion [1] and Qcluster [6], can be used in each iteration to better convey user's relevance feedback.
3. With the growing interest in Internet-scale image search applications, it is desirable to extend the target search techniques because it will enable concurrent users to share computation [7].

Experimental Results

Figure 7 shows that standard techniques MARS [9], MindReader [5], and Qcluster [6] have poor false hit ratio when *k* is small. This is due to the effect of local maximum traps. Even for fairly large *k*, their false hit ratios remain very high. As a result, users of these techniques have to examine a large number of returned images, but might still not find their intended targets. Figure 8 shows that NDC and GDC perform more efficiently when *k* is small, with GDC being slightly better than NDC. Specifically, when $k = 5$, the average numbers of iterations for LNM, NDC, and GDC are roughly 21, 10, and 7, respectively. Experimental studies based on a prototype [8] showed that only seven iterations on average were needed to locate a given target image. Additional performance results can be found in [8].

Data Sets

The data set consists of more than 68,040 images from the COREL library. Thirty-seven visual image features divided into three main groups were used: colors (9 features), texture (10 features), and edge structure (18 features).

Cross-references

▶ Content-based Image Retrieval (CBIR)
▶ Relevance Feedback
▶ Target Search

Recommended Reading

1. Chakrabarti K., Ortega-Binderberger M., Mehrotra S., and Porkaew K. Evaluating refined queries in top-k retrieval systems. IEEE Trans. knowledge and Data Eng., 16(2):256–270, 2004.
2. Cox I.J., Miller M.L., Minka T.P., Papathomas T.P., and Yianilos P.N. The Bayesian image retrieval system, PicHunter: theory, implementation, and psychophysical experiments. IEEE Trans. Image Processing, 9(1):20–37, 2000.
3. Flickner M., Sawhney H.S., Ashley J., Huang Q., Dom B., Gorkani M., Hafner J., Lee D., Petkovic D., Steele D., and Yanker P. Query by image and video content: The QBIC system. IEEE Computer, 28(9):23–32, 1995.
4. Hua K.A., Yu N., and Liu D. Query decomposition: A multiple neighborhood approach to relevance feedback processing in contentbased image retrieval. In Proc. 22nd Int. Conf. on Data Engineering, 2006.

5. Ishikawa Y., Subramanya R., and Faloutsos C. MindReader: Querying databases through multiple examples. In Proc. 24th Int. Conf. on Very Large Data Bases, 1998, pp. 218–227.
6. Kim D.-H. and Chung C.-W. Qcluster: relevance feedback using adaptive clustering for content-based image retrieval. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 599–610.
7. Liu D. and Hua K.A. Support concurrent queries in multiuser CBIR systems. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 1379–1381.
8. Liu D., Hua K.A., Vu K., and Yu N. Fast query point movement techniques with relevance feedback for content-based image retrieval. In Advances in Database Technology, In Proc. 10th Int. Conf. on Extending Database Technology, 2006, pp. 700–717.
9. Rui Y., Huang T., Ortega M., and Mehrotra S., Relevance feedback: A power tool for interactive content-based image retrieval.IEEE Trans. Circuits Syst. Video Technol., 8(5):644–655, 1998.

Query Processing

Evaggelia Pitoura
University of Ioannina, Ioannina, Greece

Synonyms

Query evaluation

Definition

A query processor receives a query, validates it, optimizes it into a procedural dataflow execution plan and executes it to obtain the results of the query.

Key Points

Query processing consists of several phases. In the first phase, the *query parser* checks whether the query is correctly specified, resolves any names and references, verifies consistency, and performs authorization tests. If the query passes validation, it is converted into an internal representation that can be easily processed by the subsequent phases. Then, the *query rewrite module*, or *rewriter*, simplifies the query and transforms it into an equivalent form by carrying out a number of optimizations that do not depend on the physical state of the system, such as view expansion and logical rewriting of predicates. These initial phases rely only on data and metadata in the system catalog.

Next, the *query optimizer* transforms the internal query representation into an efficient query plan. It decides which indices to use, which methods to use for executing the query operators and in which order. A *query plan* may be thought of as a dataflow diagram that pipes data through a graph of query operators. The optimizer enumerates alternative plans and chooses the best plan using a cost estimation model that relies on various selectivity estimation techniques and available statistics regarding the physical state of the system. The *code generation component* transforms the plan produced by the optimizer into an executable plan. Finally, the *query execution engine* operates on the fully-specified query plan. It provides generic implementations for every operator. Most execution engines are based on the *iterator model*. In such a model, operators are implemented as iterators, with all iterators having the same interface [1,2,3].

Cross-references

► Evaluation of Relational Operators
► Iterator
► Query Optimization
► Query Optimization in Sensor Networks
► Query Plan
► Selectivity Estimation

Recommended Reading

1. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73–170, 1993.
2. Haas L.M., Freytag J.C., Lohman G.M., and Pirahesh H. Extensible query processing in starburst. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1989, pp. 377–388.
3. Hellerstein J.M., Stonebraker M., and Hamilton J. Architecture of a database system. Foundations and Trends in Databases, vol. 1(2), 2007.

Query Processing (in Relational Databases)

Volker Markl
IBM Almaden Research Center, San Jose, CA, USA

Synonyms

Query compilation and execution

Definition

Query processing denotes the compilation and execution of a query specification usually expressed in a declarative database query language such as the structured query language (SQL). Query processing consists of a compile-time phase and a runtime phase. At compile-time, the query compiler translates the query specification into an executable program. This translation process (often called *query compilation*) is comprised of lexical, syntactical, and semantical analysis of the query specification as well as a query optimization and code generation phase. The code generated usually consists of physical operators for a database machine. These operators implement data access, joins, selections, projections, grouping, and aggregation. At runtime, the database engine interprets and executes the program implementing the query specification to produce the query result.

Historical Background

In the 1960s and 1970s, navigational database management systems emerged. The hierarchical database management system, IMS from IBM is a prominent example that is still widely used. The Database Task Group developed the CODASYL approach to data management during the standardization of COBOL. The CODASYL approach focuses on following pointers that link a network of data records. CODASYL lacks a declarative query language. Query processing for CODASYL databases is mainly about efficient access to data, where queries are implemented as procedural programs by software developers. There was no query optimization in the CODASYL model.

In 1970, Edgar Codd invented a relational algebra over tables. This relational data model was much more flexible than navigational data models. Relational algebra allowed the specification of arbitrary queries over relations. A combination of five primitive operations (selection, projection, union, filter, Cartesian product) over tables define a query in the relational algebra. Initially, the relational model was considered to be impractical, as it required data to be normalized into relations, and queries were very expensive operations. Queries often have to use the join operation (a combination of a Cartesian product with a selection) in order to reconstruct normalized relations. However, over the years, many inventions (database indexes, join methods like nested-loop join, merge join or hash-join, and techniques for query optimization) have together made relational database management systems feasible.

The prime example of a relational query language is SQL, which is based on a logical calculus called *tuple calculus*. An SQL query specifies *what* data to access and process in order to compute a query result, but not *how* to access and process the data. An SQL query over a relational database can be implemented in many different ways. For complex queries on databases with many indexes and tables, there may be millions of different ways to implement an SQL query. Each implementation may use different orders for combining the normalized tables, or use different join methods like hash join or nested-loop join, or use different types of indexes like B-Trees or bitmap indexes, or using indexes on different columns to efficiently process a selection over a table. Choosing the right method for processing a query over a large database can produce a query result in seconds (or faster), whereas choosing of the wrong method can result in queries running for hours, or even days.

Query optimization aims at selecting the most efficient access path (often called *query execution plan*, or *plan*) for any given query. The task of a query optimizer is to find the most efficient overall implementation of the query. Query optimization has been an active area of research since the 1970s, with advances still being made today. Some standard optimizations are based on simple heuristics. A typical objective may be to avoid large intermediate results during query processing by applying selections as early as possible. However, determining the most efficient access method for each table as well as the best join methods and join orders to combine tables cannot be carried out by simple heuristics alone. In order to find the best plan, the query processor actually needs to know some characteristics of the data that the query will process. For example, a nested-loop join method works best if one of the two tables to be joined is relatively small. A hash join has higher overhead for small tables, but will produce the query result much faster if both tables are large. Similarly, it is good to use an index if a selection results in very few rows of a table. If many rows qualify, a table scan will be much faster, as it can use sequential I/O, avoiding the consecutive repositioning of the read/write head of a hard disk as required when processing an index.

In 1979, Pat Selinger proposed a cost-based query optimizer for System R which determines the optimal query plan based on a mathematical model of the execution cost of each operator in a query execution plan. The System R optimizer, which has become the basis for many commercial database systems (e.g., IBM's DB2 and Oracle), enumerates all possible physical query execution plans for a query in a stratified bottom-up fashion. This is done by first determining the cheapest table accesses, then processing all two-table joins, three-table joins, and so on, and uses dynamic programming to prune the search space. In 1994, volcano introduced an alternative approach of a top-down optimizer based on goal-directed search and branch-and-bound pruning, which has found its way into commercial DBMS like Microsoft's SQL Server.

During the 2000s, advanced techniques for query processing have been proposed and implemented into commercial systems. One of these developments relies on feedback loops to improve query execution. Feedback obtained by monitoring some parameters of the mathematical cost model during query execution is used to either alter the query execution plan while the query is running, or to adjust the mathematical model to increase its accuracy for subsequent queries. In addition, integration of semi-structured data (XML) and the Xquery language are active areas of research, with new requirements imposed on a query processor due to the navigational operators like XPath needed for processing hierarchical data.

Foundations

The figure below gives an architectural overview of a query processor. The query compiler of a relational database translates a declarative SQL query into a procedural program. Initially, a parser carries out tokenization and creates a parse tree of the query based on the grammar of the query language. Semantical analysis then tests semantical correctness of the query. Those tests usually validate if the table names or column names in the query exist and have the right types. These parsing steps are similar to the steps a programming language parser would carry out, except that the query compiler has to generate a program for a data flow engine, not for a microprocessor. The query is therefore usually represented as a data flow graph (also called *query graph*) in a query compiler. A query graph is a graph whose edges represent the data flow and whose nodes represent operations on the data. Typical data flow operations are table access and join, group, and filter. Many query optimizers, like the optimizer of IBM's DB2 or Oracle's rule based optimizer, utilize a rule-based query rewrite phase before carrying out cost-based optimization. Query rewrite translates the query graph into a semantically equivalent query graph, which is preferable to execute. Rewrite rules include, among others, translations from subqueries into joins, or rules to generate transitive predicates to involve indexes in query processing, which would otherwise not be applicable.

The System R approach to cost-based query optimization enumerates all possible physical query execution plans for a the query graph model, associates a cost with them, and selects the cheapest plan to be executed. The cost of a query plan is expressed as a linear combination of intermediate result sizes (cardinalities) weighed by factors for CPU cost, I/O cost. Cost models for distributed systems also factor in a the communication cost for transferring data between the processing components. The optimizer uses dynamic programming and prunes the search space as early as possible by only retaining the plan with the lowest cost whenever possible. The search space can be pruned whenever some (intermediate) plans are comparable. Plans are comparable, when they are semantically equivalent, i.e., when their execution produces the same intermediate result. However, in practice, there are cases where semantically equivalent plans are not yet comparable. For example, a semantically equivalent plan may not be comparable to an intermediate plan that has an interesting property, like a particular sort order for a an intermediate result, which could be exploited at a later stage during query processing to lower the overall cost.

Plans are enumerated bottom-up in a stratified way. This stratification starts with enumerating table access plans for each table then enumerates all (intermediate) plans for combining two of these table access plans, then all (intermediate) plans for three tables, and so on, until the all plans have been enumerated that produce the overall query result. As soon as several semantically equivalent (intermediate) plans have been computed, the optimizer retains only the plan with the lowest cost for further consideration. Dynamic programming has a complexity, which is exponential in the number of tables joined in a query. If the exponential memory requirement of dynamic programming is too high due to too many joins in a query, the optimizer uses a greedy algorithm as a fallback. While dynamic programming is guaranteed to determine the optimal plan with respect to the cost model, greedy algorithms usually do not return the optimal plan. Moreover, greedy algorithms tend to have a bias towards bushy plans (i.e., balanced join trees as opposed to left-deep join trees), which usually has negative impact on pipelining and transactional queries with small intermediate results. Many commercial database systems like DB2 and Oracle use a System R style bottom-up optimizer.

Volcano's alternative approach refines the query graph model by replacing logical operators like join with physical implementations like hash-join or merge-join and uses branch and bound to limit recursion. For instance, the top-down approach is used by the query optimizer of Microsoft's SQL Server.

Both bottom-up and top-down optimizers achieve the goal of determining the optimal plan with respect to a cost model. The quality of the plan does not depend on which of these search methods is used, but rather on the repertoire of rules available for generating or expanding plans. Top-down optimizers have an advantage, though, in that they always maintain a feasible execution plan, so it is possible to stop optimization at any time and execute the currently cheapest plan. Bottom-up optimizers have the risk of running out of memory without having produced a feasible plan, thus the necessary fallback to greedy or other heuristics.

Both bottom-up and top-down optimizers need a good model for the cost of producing the intermediate results that occur during query processing. This execution cost is largely dependent upon the number of rows – often called *cardinality* – that will be processed by each operator in the plan. Typically, an estimate for the cardinality of some intermediate result relies on statistics of database characteristics. Many database systems use simple statistics to approximate the size of an intermediate result, like the number of rows for each table and as well as the number of distinct values for each column. For a simple selections with an equality predicate "$C = value$" on column C of a table with n rows and c distinct values in column C, many cost models use the simple formula $1/c * n$ to estimate the cardinality of the selection predicate. This simple formula assumes a uniform distribution of all values in column C. The cardinality estimate may be vastly incorrect if some values in column C occur more frequently than others.

Estimating the number of rows (after one or more predicates have been applied) has been the subject of much research since the 1980s. The percentage of the number of rows in a table or intermediate result satisfying a predicate P is often called *selectivity* of P. The selectivity of P effectively represents the probability that any row in the database will satisfy P. Database statistics are expensive to compute and cannot always easily be maintained incrementally, when the database changes. In general, there is a trade-off between accuracy of statistics and their storage and maintenance cost. The goal of database statistics is to be good enough to enable the optimizer to produce a robust and efficient plan. The cost model of a database systems therefore uses simplifying assumptions to compute selectivities and cardinalities from the available database statistics. Examples of these assumptions include:

1. *Currency of information*: The statistics are assumed to reflect the current state of the database, i.e., that the database characteristics are relatively stable. This may not be true, if a table is changing rapidly. In this case, statistics need to be collected frequently, using the database statistics collection tool. Outdated statistics are a major source for performance problems in database queries. Many modern database management systems have an infrastructure that tries to automatically detect when statistics are outdated by monitoring update, insert, and delete operations on a table. Those systems can be configured to (re-)collect statistics automatically when a certain amount of changes have occurred.

2. *Uniformity*: As describe above, without detailed statistics on a column, the data values within a column are assumed to be uniformly distributed. If that is not the case, the database administrator can create histograms for particular columns to deal with skew in values. This will improve the accuracy of selectivity estimation for selection predicates on a single table. Only recently have researchers begun to explore ways to improve the estimation of selectivities for join predicates which combine multiple tables.

3. *Independence of predicates*: Without any knowledge about interactions of predicates, selectivities for each predicate are calculated individually and multiplied together, even though the underlying columns may be related, e.g., by a functional dependency. This independence assumption usually results in severe underestimation of the selectivity of predicates on correlated columns. For a table containing cars with two columns, *make* and *model*, the independence assumption will result in severe estimation errors for a selection predicate like "make='VW' and model = 'Jetta'." If 10% of the cars in the table are VW, and 1% of the cars are Jetta, the independence assumption would result in a selectivity:

$$s_{\text{VW Jetta}} = s_{\text{VW}} \times s_{\text{Jetta}} = 10\% \times 1\% = 0.1\% = 0.001$$

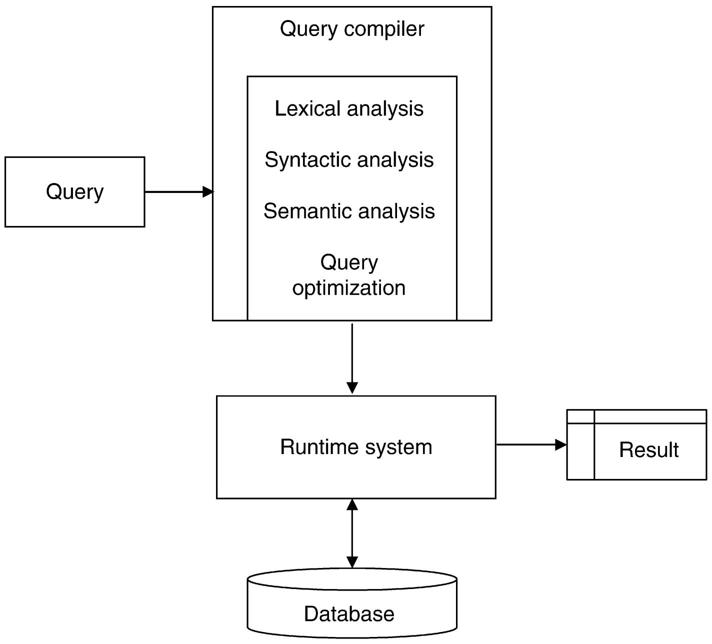
Since only VW makes Jettas, the real number is 1%, with an order of magnitude error in the estimation. This means that the intermediate result size for that part of a query will be ten times larger than what the optimizer assumes. This can result in not allocating enough memory for processing the query, or choice of a suboptimal plan. Overall, this estimation error progressively worsens, as more predicates are present in a query.

Collecting multivariate statistics across multiple columns can overcome the independence assumption when data is correlated. A simple correlation parameter is the number of distinct values over a set of attributes. This statistic is employed for instance by IBM's DB2 to correct the independence assumption for correlated local predicates, or to better estimate the selectivity of a join predicate with correlation between the two tables. While the number of distinct values over a set of columns addresses the correlation between these columns, it assumes all combinations of values in these columns to be uniformly distributed. Correlations inflate errors more than the uniformity assumption. Thus distinct values have proved worthwhile in practice as correlation statistics. In case of non-uniform correlations, multidimensional histograms have been proposed. These histograms store and maintain multivariate distribution statistics. However, multidimensional histograms do not work well for equality predicates or for maintaining correlations over a large set of columns. For that reason, they have not been widely adopted in commercial databases.

4. *Principle of inclusion:* The selectivity for a join predicate X.a = Y.b is typically defined to be 1/max{|a|, |b|}, where |b| denotes the number of distinct values of column b. This implicitly assumes the "principle of inclusion," i.e., that each value of the smaller domain has a match in the larger domain (which is frequently true for joins between foreign keys and primary keys). Again, correlation statistics can help overcome this assumption. Products like IBM's DB2 or Microsoft's SQL Server offer statistics on views, which can address incorrect assumptions about correlations of columns between tables or within tables.

Applications commonly used today have hundreds of columns in each table and thousands of tables, making it very hard for a database administrator to know on which columns to collect and maintain multivariate statistics or statistics on views. Tooling, either based on query feedback or proactive sampling, is addressing that problem by determining the most important correlated columns that joint statistics need to be collected on, employing statistics methods like χ^2-testing for correlation detection. Methods such as entropy maximization have been proposed to generalize the independence assumption and allow for dealing with arbitrary correlation between columns, either within or across tables.

Once the best overall query plan according to the optimizer's model has been determined, it is handed to the runtime system for execution. In some architectures,



Query Processing (in Relational Databases). Figure 1. An overview of query processing.

e.g., Informix, the query plan is directly interpreted by the runtime system. Other architectures follow the System R design and employ a code generator to produce code for a database machine, which is then executed by the runtime system. This additional code generation step allows for optimizations typically found in programming languages, to reduce path length and copying of data between CPU and memory (Fig. 1).

Key Applications

All major database systems (DB2, SQL Server, Oracle, Sybase, Informix, MySQL, PostGres) implement a query processor that largely follows the previously described architecture and concepts. DB2 and Oracle follow the System R style optimizer as described above, Microsoft SQL server uses a top-down optimizer.

Recommended Reading

1. Codd E.F. A relational model of data for large shared data banks. Commun. ACM., 13(6):377–387, 1970.
2. Freytag J.C., Maier D., and Vossen G. (eds.) Query processing for advanced database systems. Morgan Kaufmann, 1994.
3. Graefe G. Volcano – an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng., 6(1):120–135, 1994.
4. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73–170, 1993.
5. Lorie R.A. and Fischer N.J. An access specification language for a relational data base system. IBM J. Res. Dev., 23(3):286–298, 1979.
6. Markl V., Haas P.J., Kutsch M., Megiddo N., Srivastava U., and Tran T.M. Consistent selectivity estimation via maximum entropy. VLDB J., 16(1):55–76, 2007.
7. Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., and Price T.G. Access path selection in a relational database management system. In Proc. ACM SIGMOD Int. Conf. on Management of Data., 1979, pp. 23–34.
8. Yu C.T. and Meng W. Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, 1997.

Query Processing and Optimization in Object Relational Databases

JOHANN-CHRISTOPH FREYTAG
Humboldt University of Berlin, Germany

Synonyms

Query evaluation; Query planning and execution

Definition

In an (object) relational database management system (DBMS) query processing comprises all steps of processing a user submitted query including its execution to compute the requested result. Usually, a user query – for example a SQL query – declaratively describes *what* should be computed. Then, it is the responsibility of the DBMS to determine *how* to compute the result by generating a (procedural) query execution plan (QEP) that is semantically equivalent to the original query. Query processing also includes the execution of this generated QEP.

While generating a QEP for a user submitted query the DBMS explores a large number of potential execution alternatives. To choose the best ones among those alternatives requires one or more (query) optimization phases.

Historical Background

In the late 1960s and early 1970s it became clear that existing DBMSs were too complex and cumbersome to use. The programmer was forced to know the physical layout of the data on disc for efficient access. The answer to those problems was the Relational Model developed by E.F. Codd in the early 1970s.

With the design and implementation of relational DBMSs in the mid-1970s it became clear very quickly that query processing and optimization are the key for implementing DMBSs with acceptable performance for the end user. In 1979, the first relational DBMS, IBM's System-R laid the foundation for today's query processing and optimization approaches: In System-R, query processing consists of three major phases:

1. Syntactic and semantic checking
2. (physical) query optimization
3. Query execution

The INGRES DBMS (Stonebreaker, UC Berkeley) took a similar approach to implement query processing.

Over the last 25 years this phase-based model has been extended by

1. adding more query processing phases such as logical query optimization or query rewrite
2. considering more alternatives during query optimization to generate better QEPs

Furthermore, the complexity of query processing has increased due to more complex query languages (such as SQL-2 or SQL-3), more complex query execution environments such as parallel processors, distributed data collections, and more complex data types, richer data structuring capabilities including object orientation.

As a response to the development of object oriented database management systems (OODBMS) the Relational Model was extended with object oriented concepts and features to embrace the major concepts of those languages. The extension of the language forced the database vendors to extend query optimization and query extension to handle those new concepts correctly efficiently such as objects, classes, path expressions, inheritance, methods, and polymorphism [10]. Most of those language features are reflected in the SQL-3 Standard.

Foundations

The Four Phases of Query Processing

Today's DBMSs usually implement query processing (including query optimization) in three different phases before executing the query to generate the requested result [2,8].

During the first phase, the query is checked for syntactic and semantic correctness. Then, during the second phase the query is rewritten into a semantically equivalent one using additional (logical/conceptual) information, such as schema information (candidate keys/Primary key, uniqueness of values, foreign keys) or integrity constraints. Rewriting might also strive for a normalized form of the query to simplify the processing during the following phases [7].

Example 1:

The SQL query

```
SELECT *
FROM CUSTOMER
WHERE EXIST (
 SELECT *
 FROM ORDER
 WHERE CUSTOMER.ID = ORDER.CUSTOMER_ID
 AND ORDER.VALUE > 10.000)
```

could be rewritten into the query

```
SELECT *
FROM CUSTOMER, ORDER
WHERE CUSTOMER.ID = ORDER.CUSTOMER_ID
AND ORDER.VALUE > 10.000
```

if the inner query block returns zero or one tuple.

The third phase – commonly known as the (physical) query optimization – translates a user query into a query execution plan (QEP). While the initial query declaratively expresses which properties the result should have, the QEP determines the execution steps to evaluate the query generating the requested result.

In general, there are many different ways (QEPs) to evaluate a user submitted query based on the physical properties of the tables accessed, such as available indexes, available materialized views, sorting order, or clustering. Thus, the query optimizers must consider and evaluate many different alternatives to execute a query using different access paths and different operators.

During the QEP generation the optimizer determines

- The best (optimal) way to access each individual relation referenced in the query using indexes (including how and in which order to evaluate local predicates)
- The best (optimal) order to join two or more tables
- The best algorithm to perform the join between two tables

Example 2:

The rewritten query of Example 1

```
SELECT *
FROM CUSTOMER, ORDER
WHERE CUSTOMER.ID = ORDER.CUSTOMER_ID
AND ORDER.VALUE > 10.000
```

might be translated into the following QEP. For the sake of clarity the QEP uses a LISP-like notation to represent the relational algebra-like expression [4].

```
(OUTPUT
  (PROJECT
  (ID CUSTOMER_NAME ADDRESS)
  (NESTED_LOOP_JOIN
    (CUSTOMER.ID = ORDER.CUSTOMER_ID)
    (SCAN CUSTOMER)
    (FILTER
      (ORDER.VALUE > 10.000)
      (SCAN ORDER)))))□
```

Finally, during the last phase the DBMS executes the generated QEP to compute the requested result, i.e., the set of tuples that match the submitted query.

The Fundamentals of a (Physical) Query Optimizer

To generate the best (optimal) QEP any query optimizer embodies the following three aspects [4]:

1. A search strategy
2. (A set of) Cost functions
3. A QEP generation strategy

The search strategy determines which alternative (partial) QEPs should be generated while looking for the best (optimal) QEP among possibilities. Strategies such as exhaustive search strategies (together with dynamic programming) or heuristics (Greedy heuristic) are approaches used in current DBMSs. However it seems that exhaustive strategies – with some restrictions – are still the preferred approach to generate the best QEP despite a large search space whose size is determined by various parameters, such as the number of tables in the query, the available indexes, or the available join methods (algorithms).

While generating different alternatives for query execution, the optimizer must determine which of these alternatives is better. Therefore, cost functions assign cost values to (partially generated) QEPs by determining the resource consumption for that plan. The resources used might be CPU time, space (amount of memory), communication time (distributed query processing), and – most importantly – the number of disk IOs since disk IOs are the dominating cost in almost all QEPs. Cost functions therefore estimate ("foresee") the amount of resources that a plan will consume when being executed.

To feed the cost functions with input, most DBMSs maintain so-called *histograms* which record value distributions for various (sets of) attributes in different tables [5]. Those value distributions allow the optimizer to make informed decisions regarding the different operators of a QEP with respect to their efficiency and effectiveness.

Once the optimizer settled for one alternative during partial QEP generation it must determine how to advance the QEP generation in its search for the best (complete) QEP. This progress might be implemented by extending the current partial QEP with new operators possibly on additional tables not yet considered, or by generating alternatives for the existing QEP by transforming it into a syntactically different QEP. The latter might include considering alternative access paths (using different indexes) or replacing an existing join algorithm with a different one.

The query optimization phase terminates once the optimizers found the best QEP based on the given (searched) alternatives and based on the given cost functions. However, since the search might take time, the optimizer might also terminate after having reached a pre-determined tie limit or after having evaluated a certain number of alternatives.

Formalisms and Approaches for Query Processing and Query Optimization

The (research) literature reports on many different approaches to query processing and query optimization. However, there does not seem to be one formalism to describe those different approaches. Some of them use a relational algebra like notation (with extensions) to show operational approaches. Others use logic based notation such domain/tuple relational calculus or the tableaux notation. Some presentations use proprietary notations based on innovative data structures to present sophisticated algorithms.

Of course, the approach to query processing and query optimization is heavily influenced by the data model and the expressiveness of the query language. The continuous extension of SQL to SQL-2 or SQL-3 has lead to an extended portfolio of query processing and query optimization techniques. The same is true object oriented and object relational DBMSs where introduced. Similarly, the extension of SQL to express Data Warehouse queries leads to new techniques (algorithms and "tricks") for query processing and query optimization.

Implementing Query Optimizers

The first optimizers were implemented for the database prototypes IBM System-R [8] and Berkley's INGRES. The former system laid the foundation conceptually and architecture wise for many optimizers to come including those that are used in today's DBMS products.

In many cases, DBMS products only provide access to the architecture, the optimization strategies, or the cost functions used. Most of the dominant products allow the user to view the QEP as generated by the optimizer together with cost values and cardinality estimates.

However, for several DBMS products, there exist well known prototypes that provide hints how some of the optimizers in existing products might work. The newly researched ideas and novelties in those prototypes are often described in research papers published at well-know conferences like the ACM Sigmod conference, the VLB conference, the IEEE ICDE conference, or the European EDBT conference.

Key Applications

The query optimizer is an integral part of any DBMS (product). The degree and extent of optimization is determined by the vendors or implementers. The repertoire of the optimization step is continuously extended, depending on the requirements coming from OLTP queries, data warehousing/OLAP queries (star schema, aggregate queries), data mining queries, or queries coming from systems such as geographical information systems booking systems, and others.

Future Directions

Over the last several decades, query processing and optimization has adapted to the changes in computing hardware. In the 1990s, parallel hardware became readily available at a reasonable cost, DBMSs and query processing had to be extended and adapted to parallel QEP execution.

In addition, many database experts doubt that the general two-phase query processing approach (optimize the query, then execute) is the right one for future DBMs that should run on the next generation of hardware. Especially when the underlying system continuously changes query optimization and query execution must be closely intertwined to react to those changes in (almost) real time. Furthermore, when executing queries on large tables, it might be desirable to change execution strategies "on the fly." Currently, once the query processing phase is done (determining the best plan) the QEP is fixed. Recent research in the area of query processing has developed approaches to change QEPs either "on the fly" when necessary.

Now, multi-core CPUs seem to be the new development direction for hardware. At the same time, new storage technology comes to life (flash disks) with different operating characteristics that will change the architecture and therefore the processing models and processing capabilities of existing DBMSs. In addition, large CPU farms, large storage farms, and increasing main memory sizes already have a dramatic impact on existing approaches and techniques in query processing and query optimization. Understanding the Web as one large database could completely change today's DBMS architecture standards and assumptions on how to build future DBMSs.

Experimental Results

Unfortunately, there are no benchmarks that consider the database query optimizer as a separate component for benchmarking. However, there are efforts to ensure the stability of the optimizers despite estimation errors during query optimization and despite changes in the expected resources available during query execution.

Cross-references

- ▶ Cost Function
- ▶ Database Products that include optimizer technology
- ▶ Histogram
- ▶ IBM DB2
- ▶ Indexing
- ▶ Informix (owned by IBM)
- ▶ Ingres
- ▶ Logical Schema Design
- ▶ Materialized Views
- ▶ Microsoft SQLServer
- ▶ Oracle
- ▶ Parallel Database
- ▶ Physical schema
- ▶ Query Language
- ▶ Relational Algebra
- ▶ Relational Integrity Constraints
- ▶ Relational Model
- ▶ SAP MaxDB
- ▶ Search Strategy
- ▶ Sybase
- ▶ System-R

Recommended Reading

1. Deshpande A., Ives Z., and Raman V. Adaptive query processing. Foundations and Trends in Databases, 1(8):1–140, 2007.
2. Freytag J.C. The basic principles of query optimization in relational database management systems. In Proc. IFIP 11th World Computer Congress, 1989, pp. 801–807.
3. Freytag J.C., Maier D, and Vossen G (eds.) Query Processing for Advanced Database Systems. Morgan Kaufmann, 1994.
4. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73–170, 1993.
5. Ioannidis Y.E. The history of histograms (abridged). In Proc. 29th Int. Conf. on Very Large Data Bases, 2003, pp. 19–30.
6. Jarke M. and Koch J. Query optimization in database systems. ACM Comput. Surv., 16(2):111–152, 1984.
7. Pirahesh H., Hellerstein J.M., and Hasan W. Extensible/rule based query rewrite optimization in starburst. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1992, pp. 39–48.
8. Selinger P.G., Astrahan M.M., and Chamberlin D.D., Lorie R.A., and Price T.G. Access path selection in a relational database

management system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 23–34.
9. Yu C.T. and Meng W. Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, 1998.

Query Processing in Data Warehouses

Wolfgang Lehner
Dresden University of Technology, Dresden, Germany

Synonyms

Data warehouse query processing; Query answering in analytical domains; Query optimization for multidimensional systems; Query execution in star/snowflake schemas

Definition

Data warehouses usually store a tremendous amount of data, which is advantageous and yet challenging at the same time, since the particular querying/updating/modeling characteristics make query processing rather difficult due to the high number of degrees of freedom.

Typical data warehouse queries are usually generated by *on-line analytical processing* (OLAP) or *data mining* software components. They show an extremely complex structure and usually address a large number of rows of the underlying database. For example, consider the following query: ‘Compute the monthly variation in the behavior of seasonal sales for all European countries but restrict the calculations to stores with > 1 million turnover in the same period of the last year and incorporate only 10% of all products with more than 8% market share. ‘In a first step, this query has to identify certain stores based on the previous year’s sales statistics and it needs to define the top-selling products on a monthly basis. In a second step, the system is able to compute the different behavior based on individual countries.

The structural complexity as well as the huge data volume addressed by data warehouse queries makes it a true challenge to optimize and execute these queries. Therefore, query processing in data warehouse systems encompasses the definition, the logical and physical optimization, as well as the efficient execution of analytical queries. The specific techniques range from classic rewrite rules to rules considering the special structure of a query. For example, data warehouse queries usually target *star schemas*. Such star queries with references to a fact table and foreign-key joins to multiple dimension tables can be beneficially exploited during query optimization and query processing. Moreover, the embedding of specialized operators (like *CUBE* or ROLLUP) or approximate query processing methods using samples or general-purpose synopses has to be addressed in the context of query processing in data warehouse environments. Finally, query processing in data warehouse systems has to address the support of non-*SQL* query languages (e.g., specialized analytical query languages like MDX [13]), alternative storage structures (e.g., column-wise organization), or alternative processing models based purely on main-memory structures.

Historical Background

Query processing has a long history with research and commercial products. The generation of efficient plans for the smooth execution of database queries thus represents one of the most challenging research directions in database history. With the advent of descriptive query languages, this has turned into a crucial problem, since systems must now figure out the optimal way to execute queries.

Traditional *transactional*-style (on-line transaction processing = OLTP) interaction patterns between an application and the database system consist of short read and write queries with correlated sub-queries, which typically compete with a large number of concurrently running transactions. DWH-style query patterns follow a significantly different style. Since the update/load of data warehouse databases is under the control of data warehouse monitors, user queries are typically read queries that touch a large number of rows with star joins and group-by operations as their functional core. Around 1995, the concept of data warehousing emerged and showed that OLTP-style query processing and optimization techniques were insufficient in dealing with this specific class of requirements. This deficit triggered a tremendous amount of research activities to push the envelope in multiple directions. After more than 10 years of research and development, a fairly large number of extensions/modifications have turned it into a commercially available system. However, due to the increasing need for analytical tasks performed within (and not on top of) database systems, the improvement of query processing techniques for analytical applications still represents an active research area with significant potential for innovation.

Foundations

Query processing in data warehouse systems does not address individual methods or techniques. In contrast, query processing in the analytical domain is characterized by a broad range of individual techniques ([3,7]; see 3.13). The challenge in this context is to orchestrate the different and individually deployed methods. The following introduction to important query processing aspects, including a brief discussion of the requirements and specifics of query processing in data warehouse systems and possible solutions, provides a comprehensive overview of the topic.

Requirements and Specifics of the Analytical Context

While the specific character of data warehouse systems poses challenges for query processing from multiple perspectives, it also has constraints, which can be exploited and therefore reduce the number of space limitations. Although the processing and optimization of *SQL* queries attracts the most attention, query processing in data warehouse environments addresses other query languages like MDX [13], DMX [5], or SQL/XML [12] with analytical functionality specified within the *XPath/XQuery* fragments. The complexity of handling domain-specific query languages and of mapping these expressions either to highly specialized storage structures or to SQL is further increased by the need for support of domain-specific operators. In this context, the integration of data mining algorithms for cluster searches or for the computation of association rules into database systems can be seen as a prominent example.

To counterbalance these requirements, query processing in data warehouse systems shows a number of limitations or specializations that facilitate adequate solutions. For example, a high ratio of analytical queries is issued to compute standardized reports (e.g., legal reporting in the financial sector, cockpit solutions, etc.). These canned queries show huge potential to pre-compile the execution plans, to apply multiple-query optimization or to pre-compute partial results using materialized views. Furthermore, the explicitly controlled update transactions in data warehouse systems typically show append-only characteristics, which enable the system to apply specifically tailored optimization strategies. For example, adding incoming rows to a single partition and periodically attaching it to the analytical database in an atomic way affects the concurrently running analytical queries only to a minimal extent. Finally, although analytical queries usually apply aggregation techniques over a large number of rows, the analytical queries may exhibit quite selective predicates. For example, consider the following query: ‘Return the weighted sales distribution of all Californian stores selling Mac and Linux machines with more than 4GB RAM in 2007 and divide it by month.’ It shows a highly selective predicate but may still incorporate a huge number of rows within the computation of the measure. Query processing may tackle this phenomenon by introducing special index structures to support selective predicates of multiple columns from different tables or by applying specialized processing techniques such as online aggregation or approximate query answering in general.

Potential and Solutions for Efficient Data Warehouse Query Processing

The specific characteristics as well as the specific circumstances create a wide range of options to perform query processing in data warehouse scenarios in an efficient way. The following list provides the most prominent classes of methods with an outline of their specific role in this context:

Part I: Query Planning and Execution

Optimizing the Optimization Process

- Since analytical queries usually show a complex structure, the optimization of a query may require a considerable fraction of the overall query execution cost. Therefore, modern techniques propose a variety of solutions ranging from the a-priori limitation of the search space (usually by applying certain heuristics – e.g., the optimization pattern of star queries) to the pruning of alternative plans as early as possible [15].

Optimization Goal

- In contrast to OLTP-style query optimization, the general goal of OLAP-style query optimization is to generate only a good (i.e., not an optimal) but robust plan. Due to the complex structure of a query, potential errors in the cost estimation usually have their origin at the leaf nodes of an operator tree and – after having been propagated along an operator path – they may result in extremely vague estimates. A robust plan is required to expose well-performing behavior even if the data show different characteristics than those used in the planning phase. The notion of robust plans and the design of more adequate statistics (e.g., sample or wavelet synopses) are currently the subject of intense discussions in the research community.

Specific Rewrite Rules

- As already mentioned, data warehouse environments imply a specific characteristic of queries – usually following the notion of star queries, which consist of a join of the fact table and multiple dimension tables followed by a complex selection predicate and a grouping condition with aggregation. Query processing has to detect such situations and apply specific optimization patterns. For example, group-by operations can be pushed down in certain cases to reduce the number of rows for join operations [18]. Another example addresses star joins: the star join technique may decide to compute the Cartesian product of the selected parts of all dimension tables in a first step and then join the (large) fact table in a last step.

Multi-Query Optimization

- A final class of techniques – which have not been considered in OLTP-style query processing due to restrictions of *ACID properties* – address the issue of optimizing a set of queries simultaneously, thus resulting in shared use of different resources. Key applications for these techniques can be found on the application level (in the computation of multiple similarly structured statistical reports) or within a system (in the propagation of changes of a base table to multiple dependent materialized views to share update efforts).

Part II: Considering Logical Access Paths

Partition Management

- Partitions reflect a concept that is useful for the administration of a database system as well as for the optimized execution of partition-aware queries. Within a data warehouse environment, incoming data items are typically stored within staging tables. After transformation and cleaning steps, tables are converted into partitions and attached to the global data warehouse database. Whenever data items in a partition are obsolete, a partition can be detached and moved (as a table) to the archive. From a query optimization perspective, partition pruning represents a powerful mechanism to potentially reduce the number of rows accessed within a query. Whenever a selection predicate refers to a partitioning criterion, the system may restrict the query execution to only those partitions that are actually referenced by the query. The main partitioning criterion usually addresses the time dimension, e.g., by month, but may also be used to partition other dimensions, e.g., by product category or geographical entities.

Materialized Views

- A second component of the logical access path consideration is the transparent use and the implicit maintenance of materialized views. Similar to classic index structures, a query is matched against a view description and internally rewritten to exploit an existing materialized view by producing the same result as the original query. Especially in the context of aggregation queries, the concept of materialized views represents an extremely powerful mechanism to speed up analytical queries. For example, a materialized view may hold summary data grouped on the family level within a product dimension, with an additional grouping based on city and month. Every query with a compatible aggregation function and a grouping condition that is “coarser” than the grouping combination of the materialized view, e.g., sum per quarter, product category, and state, may benefit from the pre-aggregated data stored within a materialized view. In addition (and similar to physical index structures), materialized views are transparently maintained in the case of changing base tables to provide a consistent view of the data.

Part III: Considering Physical Access Paths

Data Organization

- While classic relational database engines favor the concept of row-based storage, a variety of specialized systems exist that follow the concept of column-based storage layout. Query processing in column-based systems benefits from reading only those columns that are actually required to answer an incoming query. This feature is extremely beneficial in the presence of wide dimension tables and queries referring to just a few attributes.

Compression and Main-Memory Techniques (see *main memory DBMS*)

- Although (persistent) storage costs have been decreasing significantly, a compressed version of data may speed up the query processing because of the reduced number of I/O operations. While block- and table-based compression schemes have been well understood, the static characteristics of data warehouse data have provided the motivation to push compression techniques into commercial systems. For column-based systems with the primary goal to be main-memory-centric, compression is a must; a large variety of techniques (e.g., Huffman coding schemes) is applied in different systems.

Additional Index Structures

- The history of database research has shown that different application areas can be supported with specialized index structures. Particularly aimed at the support of queries within the analytical domain, *bitmap index* structures have experienced a rejuvenation and are now part of the prominent commercially available systems [2,16]. Another approach to take advantage of the static characteristic of a data warehouse database can be seen in the concept of *star index*es (also called join indexes or foreign-table indexes), which follow the basic idea to index the result of a join operation. Join indexes are particularly well-suited to exploit the relationship of dimension tables with the corresponding fact table.

Multidimensional Clustering Schemes

- *Clustering* in general tries to preserve the topological relationship of entries in a database. Since data warehouse datasets are typically multidimensional in nature (e.g., sales by shop, date, and product), multidimensional clustering schemes represent a valuable solution to store logically related facts within the same block with the overall goal to reduce the number of I/O operations.

Part IV: Alternative Query Answering Models

Online Aggregation

- Analytical queries typically show a response time in the range of multiple seconds, minutes, and sometimes hours. Online aggregation [9] is a promising technique to return intermediate results of the queries while the query is still running; the intermediate results are refined step by step until the final result is computed or until the application is satisfied with the precision of the answer.

Approximate Query Answering/Approximate Query Processing

- Many application areas that perform statistical analyses prefer to yield an approximate but quick answer instead of delaying an exact answer. The concept of approximate query answering addresses this idea and proposes techniques for the online or offline design of database synopses, which are then used to answer incoming queries. The design of synopses ranges from simple uniform samples over specifically tailored (stratified) samples to wavelet synopses. The main challenge of approximate query answering consists of the creation and exploitation of synopses on the fly (i.e., within the context of the execution of a single query) and of the provision of error bounds to derive a quality measure to be returned to the user.

This list of methods and techniques outlines the most influential factors for query processing in data warehouse environments. Behind every perspective touched in this description, there are huge numbers of specific research issues – solved and unsolved ones.

Key Applications

Over the last 10 years, database systems have become the foundation of every larger analytical infrastructure (usually embedded within a data warehouse system). Therefore, all analytically-flavored applications heavily exploit specialized database support. This obviously large spectrum of applications ranges from mass reporting for the computation of thousands of parameterized reports to the support of interactive data cube exploration (OLAP). As data mining methods are becoming more and more standardized techniques, the computation of association rules, classification trees, and (hierarchical) clusters represent key applications.

Future Directions

In the future, query processing in analytical domains will be driven by two factors. On the one hand, data volumes will continue to grow significantly, mainly because of advances in data integration efforts (to ease the pain of manual integration of additional data sources) and the growing presence of sensors to track individual items. More data will go hand in hand with the presence of increased main-memory capacity and the integration of solid-state disks (SSD) into the memory hierarchy. Physical database design and the corresponding exploitation of specialized structures will be major challenges.

On the other hand, the topic of query processing will be faced with more sophisticated statistical methods, which have to be natively supported by the database engine to yield reasonable query performance. While for many analytical application infrastructures, the "bring the data to the statistical package" method still holds, this approach will be inverted at some point. Statistical packages will work more closely with database systems, implying that computational components are brought closer to the data (as an integral part of a database engine) to be integrated within the overall optimization process. Also, the scope of analytical applications will widen and incorporate comprehensive *data visualization* techniques.

Both future developments on the application level will have significant consequences for the design of query processing techniques ranging from low-level data organization up to the support of a specialized query interface.

Cross-references

- ▶ Approximate Query Processing
- ▶ Bitmap Index
- ▶ Cube
- ▶ Data Sampling
- ▶ Main-Memory DBMS
- ▶ Multidimensional Modeling
- ▶ Parallel Query Processing
- ▶ Query Optimization
- ▶ Query Processing
- ▶ Query Rewriting Using Views
- ▶ Snowflake Schema
- ▶ Star Schema
- ▶ SQL

Recommended Reading

1. Celko J. Joe Celko's Data Warehouse and Analytic Queries in SQL. Morgan Kaufmann, 2006.
2. Chan C.-Y. Bitmap index design and evaluation. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998, pp. 355–366.
3. Chaudhuri S.and Dayal U. An Overview of Data Warehousing and OLAP Technology. ACM SIGMOD Rec., 26(1): 65–74, 1997.
4. Clement T.Y. and Meng W. Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, 1997.
5. Data Mining Extensions (DMX) Reference. Available at: http://msdn2.microsoft.com/en-us/library/ms132058.aspx
6. Graefe G. Query Evaluation Techniques for Large Databases. In ACM Comput. Surv., 25(2), 1993, S. 73–170.
7. Gray J. et al. The Lowell Database Research Self Assessment, June 2003. Available at: http://research.microsoft.com/~gray/lowell/
8. Gupta A. and Mumick I. Materialized Views: Techniques, Implementations and Applications. MIT Press, Cambridge, MA, 1999.
9. Hellerstein J.M., Haas P.J., and Wang H.J. Online Aggregation. In Proc. ACM SIGMOD Int. Conf. on Management of Data,1997, pp. 171–182.
10. Inmon W.H. Building the Data Warehouse. 2nd edn, Wiley, NY, USA.
11. Niemiec R. Oracle Database 10g Performance Tuning Tips & Techniques, 2007.
12. N.N. ISO/IEC 9075–14:2003: Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML). Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber = 35341.
13. N.N. Multidimensional Expressions (MDX) Reference. Available at: http://msdn2.microsoft.com/en-us/library/ms145506.aspx.
14. Roussopoulos N. The logical access path schema of a database. In IEEE Trans. Softw. Eng., 8, (6)S.563–573,1982.
15. Tao Y., Zhu Q., Zuzarte C., and Lau W. Optimizing large star-schema queries with snowflakes via heuristic-based query rewriting. In Proc. Conf. of the IBM Centre for Advanced Studies on Collaborative Research, 2003, pp. 279–293.
16. Valduriez P. Join indices. ACM Trans. Database Syst., 12(2):218–246, 1987.
17. Weininger A. Efficient execution of joins in a star schema. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2002, pp. 542–545.
18. Weipeng P.Y. and Larson, P. Eager Aggregation and Lazy Aggregation. In Proc. 21th Int. Conf. on Very Large Data Bases, 1995, pp. 345–357.

Query Processing in Deductive Databases

LETIZIA TANCA
Politecnico di Milano University, Milan, Italy

Synonyms

Datalog query processing and optimization; Recursive query evaluation; Logical query processing and optimization

Definition

Most of the research work on deductive databases has concerned the *Datalog language*, a query language based on the logic programming paradigm which was designed and intensively studied for about a decade. Its origins date back to the beginning of logic programming, but it became prominent as a separate area around 1978, when Hervé Gallaire and Jack Minker organized a workshop on logic and databases. In this entry, the definition of the typical computation styles of Datalog will be given, the most important optimization types will be summarized, and some developments will be outlined.

Historical Background

The research on deductive databases was concentrated mostly between the mid-1980's and the mid-1990's. In those years, substantial efforts were made to merge Artificial Intelligence technologies with those of the Database area, with the aim of building *large and persistent Knowledge Bases*. An important contribution towards this goal came from database theory, which concentrated on the formalization of Datalog – specifically designed for the logic-based interaction with large knowledge bases – and on the definition of computation and optimization methods for Datalog rules [2,5,7,14]. In parallel, various experimental projects showed the feasibility of Datalog as a data-oriented logic programming environment [4,13].

The reaction of the database community to Datalog has often been marked by skepticism. In particular, the immediate practical use of research on sophisticated rule-based interaction has often been questioned. However, the research experience on Datalog, properly filtered, has taught important lessons to Database researchers, setting the basis for the theoretical systematization of several related issues [1].

Foundations

Datalog is in many respects a simplification of the more general *Logic Programming* paradigm [11], where a program is a finite set of *facts* and *rules*. Facts are assertions about the reality of interest, like *John is a child of Harry*, while rules are sentences which allow deducing of facts from other facts. For instance, a rule might say *If X is a child of Y, and Y is a child of Z, then X is a grandchild of Z*. Facts and rules may contain variables; facts that only contain constants are called *ground facts*.

In the formalism of Datalog, both facts and rules are represented as *Horn clauses*:

$$L_0 : -L_1, \ldots, L_n$$

where each L_i is a *literal* of the form $p(t_1 \ldots t_n)$, such that p is a *predicate symbol* and the t_i are *terms*. A term is either a constant or a variable, while functional symbols are not allowed, at least in the basic syntax of Datalog. The left hand side (LHS) of a Datalog clause is also called its *head*, while the right hand side (RHS) is its *body*. Clauses with an empty body are the facts: indeed, the body contains the clause premises, thus if there are no premises this means that the head is an assertion. Clauses with a non-empty body are the rules. A set of ground facts can easily be thought of as a relational database, since each fact *parent (John, Harry)* (between John and Harry there is a child-parent relationship) can be written as a tuple ⟨*John, Harry*⟩ stored into a relation PARENT. Both facts and rules are a form of *knowledge*; indeed, the knowledge stored in the database (alternatively represented as the set of ground facts) is enriched by the knowledge which can be deduced from the rules.

In the context of general Logic Programming, it is usually assumed that all application-relevant knowledge (facts and rules) are contained within a single logic program. On the other hand, Datalog has been developed for applications which use large, relational databases; therefore two sets of clauses will be considered: the set of ground facts, called *Extensional DataBase (EDB)*, and the set of rules, i.e., the Datalog program, called the *Intensional DataBase (IDB)*. The *Herbrand Base* is the set of all ground facts that can be expressed in the language of Datalog, by using all the constants present in the database and all the predicates of $EDB \cup IDB$.

A *Datalog program* is a set of ground facts (possibly stored into a relational database) and rules, satisfying the safety condition that *all the variables contained in the rule head must also be present in its body*.

Note that a Datalog program can be considered as the specification of a *query against the EDB*, producing as answer the (set of) relation(s) of the IDB. It often happens that a user is interested in a subset of the (large) relation(s) that can be defined from a Datalog program: a *goal* is a single literal, preceded by a question mark and a dash, used to express constraints on the relations specified by the Datalog program. For example, ? − *parent*(*John*,*X*). specifies all the *X* such that *John* is a child of *X*.

Consider, as an example, the EDB constituted by a unary relation PERSON and a binary relation PARENT (containing all the pairs ⟨*child*,*parent*⟩) and the following program:

$$r_1:\ sgc(X,X):-person(X).$$

$$r_2:\ sgc(X,Y):-parent(X,X1),sgc(X1,Y1),\\ parent(Y,Y1).$$

Rule r_1 simply states that a person is a cousin at the same generation of him/herself, while rule r_2 says that two persons are same generation cousins if they have parents who, in turn, are same generation cousins. Note that r_2 is *recursive*, and that rule r_1 constitutes the recursion base. A typical goal against the set $\{r_1,\ r_2\}$ is ? − *sgc*(*John*, *Y*)., asking for all the same generation cousins of John.

Evaluation of Datalog Programs

Consider a Datalog rule $R = L_0 : -L_1,...,L_n$, and a set of ground facts $F = \{F_1,...F_n\}$. If a substitution θ exists, such that $\forall 1 \le i \le n$, $L_i\theta = F_i$, then, from the rule *R* and from the facts *F*, the fact $L_0\theta$ can be inferred in one step. This fact might be new, or already known.

The just described inference rule – which is actually a meta-rule – is called *EPP (Elementary Production Principle)*. Consider now a whole Datalog program *S*. New knowledge can be obtained from the Datalog program by applying it to the set of all ground facts of the database, to obtain new ground facts. Informally, it can be said that a ground fact *F* is *inferred* from *S* ($S \vdash F$) iff, either *a)* $F \in S$ or *b)* *F* can be obtained by applying the EPP *a finite number of times*. More precisely:

- $S \vdash F$ if $F \in S$.
- $S \vdash F$ if a rule *R* and ground facts $F_1,...,F_n$ exist such that $\forall 1 \le i \le n$, $S \vdash F_i$, and F_i can be inferred in one step by the application of *EPP* to *R* and $F_1,...,F_n$.

The sequence of applications of *EPP* which is used to infer a ground fact *F* from *S* is called a *proof* of *F* from *S*. The *proof-theoretical framework* thus established allows to infer new ground facts from an original set of Datalog clauses; on the other hand, there is a model-theoretic approach, which provides a definition of *logical consequence* (⊨). It is possible to prove that:

Theorem 1. (Soundness and completeness of Datalog) Let *S* be a set of Datalog clauses, and let *F* be a ground fact. Then $S \vdash F$ if and only if $S \vDash F$.

A proof of this theorem can be found, for example, in [5].

In order to check whether EPP applies to a rule $R : L_0 : -L_1,...,L_n$ and to an ordered list of ground facts $F_1,...,F_n$, an appropriate substitution θ for the variables of *R* must be found, such that $\forall 1 \le i \le n$, $L_i\theta = F_i$.

Given a finite set *S* of Datalog clauses, i.e., a Datalog program, according to the soundness and completeness theorem, the set of all facts which are *derivable* from *S* is the set *cons(S)* of the *logical consequences* of *S*, which can be computed by the following algorithm:

```
FUNCTION INFER(S)
INPUT: a finite set S of Datalog clauses
OUTPUT: cons(S)
begin
W:= S
while EPP produces a new ground fact F ∉ W
do W := W ∪ F
return (facts(W))
end
```

The *INFER* algorithm always terminates and produces as output a *finite* set of ground facts, *cons(S)*, since the number of constants and predicates symbols, as well as the number of arguments of these predicates, is finite. The order in which *INFER* generates new facts corresponds to the *bottom-up* order of a proof tree, thus the principle underlying *INFER* is called *bottom-up evaluation*, or, as in Artificial Intelligence, *forward chaining* (forward in the sense of the logical implication contained in the Datalog rules).

The set *cons(S)* can also be characterized as *the least fixpoint* of the transformation T_S, a mapping from 2^{HB} to itself defined as follows:

$$\forall W \in HB,\ T_S(W) = W \cup FACTS(S)\\ \cup INFER1(RULES(S) \cup W)$$

where *INFER1(S)* denotes the set of *all* ground facts that can be inferred in one step from S via *EPP*.

Accordingly, *cons(S)* can be computed by fixpoint iteration, i.e., by computing, in order, $T_S(\emptyset)$,$T_S(T_S(\emptyset)$, $T_S(T_S(T_S(\emptyset)))$,...,until a term which is equal to its predecessor is reached. This final term is *cons(S)*.

The *top-down* evaluation of a Datalog program is based on a radically different approach, where proof trees are built from the top to the bottom, by applying *EPP* "backwards," which is much more appropriate when a goal is specified together with *S*. The general principle of *backward chaining* corresponds, in Prolog, to the *SLD resolution (SL resolution with Definite clauses)* inference rule, introduced by Robert Kowalski [9]. Its name is derived from SL resolution, which is both sound and refutation complete for the unrestricted clausal form of logic.

The logic programming formalism is now related to a database query language, in order to show how easy the integration between the two realms is. Each clause of a Datalog program can be translated into an inclusion relationship of Relational Algebra; then, the set of relationships which refer to the same predicate are interpreted as Relational Algebra *equations*, whose constants are the EDB relations and whose variables are the IDB predicates, defining virtual relations. Determining a solution of the thus composed system corresponds to determining the values of the variable relations, i.e., to finding the ground facts in the IDB predicates. Consider a Datalog clause:

$$C : p(\alpha_1 ... \alpha_n) : -q_1\left(\beta_{k_1} ... \beta_{k_h}\right), ... q_m\left(\beta_{k_j} ... \beta_{k_m}\right)$$

the translation associates to C an inclusion relationship $Expr(Q_1,...,Q_m) \subseteq P$ among the relations $Q_1,...,Q_m,P$ that correspond to the predicates $q_1,...,q_m,p$, adopting the convention that relation attributes are named according to the number of the corresponding argument of the related predicate. For example, the Datalog rules of the *same generation cousins* program are translated into the inclusion relationships:

$$\pi_{1,1} PERSON \subseteq SGC$$

$$\pi_{1,5}\left((PARENT \bowtie_{2=1} SGC) \bowtie_{4=2} PARENT\right) \subseteq SGC$$

The rationale behind this translation is that literals with common variables correspond to equi-joins over those variables, while the variables exported to the rule head correspond to projections. The new (virtual) relation SGC (actually, a *view*) is defined as the extension of the predicate *sgc*. For each IDB predicate, all the related inclusion relationships are collected, generating an algebraic equation that obtains the predicate by performing the union:

$$SGC = \pi_{1,1} PERSON \cup \pi_{1,5}((PARENT \bowtie_{2=1} SGC) \bowtie_{4=2} PARENT)$$

Logical goals are also translated into algebraic queries, over the EDB or over the just defined views: $? - sgc(John, Y)$. corresponds to $\sigma_{1=\text{“}John\text{”}} SGC$.

Note that this translation is based on the use of all classical algebraic operators, except *the difference*. In fact, it can be shown that Datalog without recursion is equivalent to Relational Algebra deprived of the difference operator.

Optimization of Datalog Programs

The evaluation of Datalog programs according to various forms of fixpoint computation, similar to the INFER algorithm, is called *naive*, as opposed to better, more performant techniques whose mutual relationship is not always obvious. Optimization methods can be observed with regard to different orthogonal dimensions: the *formalism* (logical vs. algebraic), the *search strategy* (bottom-up vs. top-down), the *technique* (rewriting programs into more efficient ones vs. directly applying an efficient evaluation method), and the *type of information exploited by the optimization process* (semantic vs. syntactic).

Since Datalog programs can equivalently be written as sets of algebraic equations, a Datalog program can actually be evaluated in the same way as any algebraic query, provided that a way to process recursion is available. Actually, algebraic evaluation methods that mimick the naive method have been introduced, and the classical results of algebraic query optimization, like common subexpression analysis and equivalence transformations, have been profitably transformed to be applied to recursive queries [5].

As far as the search strategy is concerned, observe that bottom-up methods actually consider rules as *productions*, generating all possible consequences of $EDB \cup IDB$ until no new facts can be deduced; thus, these methods are applied in a set-oriented fashion, which is a desirable feature in the database context, where large amounts of data are stored in mass memory and must be retrieved in the buffer in a "set-at-a-time" way. On the other hand, also observe that bottom-up methods do not take in immediate advantage the selectivity due to the existence of bound arguments in the goal predicate.

By contrast, top-down methods [5,9] use rules as *subproblem generators*, since each goal is considered as a problem to be solved. The initial goal $? - p(\alpha_1 ... \alpha_n)$. is matched against some rule $C : p(\alpha_1 ... \alpha_n) : -q_1\left(\beta_{k_1} ... \beta_{k_h}\right), ... q_m\left(\beta_{k_j} ... \beta_{k_m}\right)$, and generates *subgoals* $? - q_i(\beta_1 ... \beta_i)$ that represent new subproblems to be solved. In this case, if the goal contains some bound (i.e., constant) argument, then only facts that are related to the goal constants are involved in the computation. Suppose, for instance, that the goal ? − $sgc(John, Y)$. be given. Then, when applying top-down rule r_2, the subgoal $parent(X, X1)$ is only further analyzed with regard to the parents of John, that is, *only the subrelation* $\sigma_{1\,=\,\text{“John”}}PARENT$ is involved in the computation of the first literal. However, although the algorithms based on this evaluation method already produce some optimization, they may be inappropriate for the database context, because many of them work "one-tuple-at-a-time."

Another analysis dimension for optimization methods is whether they directly interpret the program or first rewrite it into an equivalent, more efficient form and then evaluate it in a naive way. To this category belong, for instance, the *Magic Sets* and *Counting* methods [2], where the authors "simulate" the binding propagation achieved by top-down evaluation by applying a simple program transformation to a certain class of programs, in a similar way as algebraic database optimization techniques (e.g., push of the selections) are applied. A similar approach, directly introduced by means of the algebraic formalism, is taken in the *Reduction of Variables* and *reduction of Constants* methods [5].

Finally, also *semantic information* can be used to optimize programs: for instance, [6] base the optimization on the additional semantic knowledge provided by *database constraints*. For example, a constraint might state that *all sailing vessels in Ischia are sheltered in the main harbour*, thus the query asking for *the harbour in Ischia where the sailing boat "Roxanne" is located* can be answered without even accessing the DB.

Negation in Datalog

In pure Datalog, the negation sign ¬ is not allowed; however, negative facts can be inferred from Datalog programs by adopting the *Closed World Assumption (CWA)*, which, in this context, reads as follows:

If a fact is not derivable from a set of Datalog clauses, then the negation of that fact is true.

For example if, after computing the SGC relation, the tuple ⟨*LUCY, JOHN*⟩ is not found, the fact $\neg sgc\,(lucy, john)$ – that is, Lucy is not a child of John – is inferred.

The CWA applied to Datalog clauses allows the deduction of negative facts, but not their use within the Datalog rules in order to deduce some new facts. For instance, in Relational Algebra, all the pairs of persons who are not same generation cousins are specified as: $NONSGC = (PERSON \times PERSON) - SGC$. Accordingly, one would like to write:

$$r_3: \; nonsgc(X, Y) : -\, person(X), person(Y), \neg sgc(X, Y).$$

The language *Datalog*$^{\neg}$ has the same syntax as Datalog, but here negated literals are allowed in the rule bodies. For safety reasons, all the variables occurring within a negated literal must also occur in a positive literal of the body. Without delving into semantic details, note that unfortunately, because of recursion, the computation of Datalog$^{\neg}$ programs is not as straightforward as that of a pure Datalog program; indeed, by simply applying the *EPP* and the INFER algorithm to a recursive Datalog$^{\neg}$ program, one may incur into a contradiction, since some negative facts that are inferred at step n of the computation might be derived as positive at some step $n + k$. Intuitively, consider the program composed by rules r_1, r_2, r_3, and think of the set of same generation cousins derived at the first computation step. Suppose that $sgc(lucy, john)$ has not been derived; then, at the second step, $\neg sgc(lucy, john)$ may be derived by applying r_3; yet, it might be that $sgc(lucy, john)$ is derived from r_1, r_2 at some later step $k(k > 2)$.

The most common policy used to avoid such problems is only allowing Datalog$^{\neg}$ programs which are *stratified* [7], according to the following intuition: when evaluationg a rule with one or more negative literals in the body, *evaluate first the predicates corresponding to these negative literals*. Then, the CWA is "locally" applied to these predicates. Consider the example above: since the *sgc* predicate appears in negative form in rule r_3, it is first evaluated by applying only rules r_1,r_2 (the "first stratum" of the program); then, once the whole extension of *sgc* has been computed, all the pairs of persons who are not in the SGC relation can be derived (actually by difference).

However, one can intuitively guess that not necessarily all the Datalog programs can be stratified in this way, that is, it may be possible that there is another rule r_4 which in turn contains the predicate *nonsgc* in negative form, and so on and so forth. More formally, define the *Dependency Graph* $DG(P) = \langle N(P), E(P)\rangle$ of a Datalog$^{\neg}$ program P as follows: $N(P)$ is the set of predicates p occurring in the rule heads of P, while an edge $\langle p,q \rangle$ belongs to $E(P)$ if the predicate symbol q occurs positively or negatively in some rule whose head predicate is p. Moreover, $\langle p,q \rangle$ is labeled by $\neg$ if there is at least one rule in P with head predicate p, whose body contains a negative occurrence of q.

A Datalog$^{\neg}$ program P is *stratified* iff DG(P) does not contain any cycle involving an edge labeled by$\neg$. If P is stratified, it is quite easy to construct a stratification of P [1,5,14], that is, a sequence of subprograms $P_0 = EDB$, $P_1,\ldots,P_n$ of P such that $P_0 \cup P_1 \cup \ldots \cup P_n = P$ and, by evaluating them separately and in order from P_0 to P_n, and by applying the CWA to P_k when computing P_{k+1}, the result does not contain any contradiction. Note that stratifications are not unique, that is, if the condition on the $DG(P)$ is satisfied, P can be stratified in several different ways. It is easy to see that the sample program $P = \{r_1,r_2,r_3\}$ is stratified, and that a stratification is $P_0 = EDB$, $P_1 = \{r_1,r_2\}$, $P_2 = \{r_3\}$.

Much further work has been done on the semantics of negation and of non-monotonic programs [1]. Semantics based on various kinds of partial models, like the *stable models* [12] and the *well-founded models* [10] are often studied. An example of more recent work on the subject is [3], where the new concept of soft stratification, based on a new bottom-up query evaluation method based on the Magic Set approach, is proposed.

Key Applications

The research on deductive databases was concentrated in the decade between the mid-eighties and the mid-nineties. This work, and all logic-based approaches to database problems, constitute the foundational experience for speculation that have lead to a number of results in different fields. The field of active databases is one interesting example of the application of the theoretical foundations of Datalog. An active database system is a DBMS endowed with active rules, i.e., stored procedures activated by the system when specific events occur. The processing of active rules is characterized by two important properties: termination and confluence. In [8], a set of active rules is translated into logical clauses, taking into account the system's execution semantics, and simple results about termination and determinism available in the literature for deductive rules are transferred to the active evaluation process. Another, more recent application is the integration of Databases with the Semantic Web, an interesting example of which is the *SWRL (Semantic Web Rule Language)* [15], a proposal combining sublanguages of the OWL Web Ontology Language with the rule-based paradigm. SWRL is (roughly) the union of Horn logic and OWL, where rules are of the form of an implication between an antecedent (body) and consequent (head), with a semantics very similar to that of Datalog.

Cross-references

▶ Active Databases
▶ Logics and Databases
▶ Query Language
▶ Semantic Web and Ontology
▶ Views

Recommended Reading

1. Abiteboul S., Hull R., and Vianu V. Foundations of Databases. Addison-Wesley, 1995.
2. Bancilhon F., Maier D., Sagiv Y., and Ullman J.D. Magic sets and other strange ways to implement logic programs. In Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1986, pp. 1–15.
3. Behrend A., Soft stratification for magic set based query evaluation in deductive databases. In Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2003, pp. 102–110.
4. Bocca J.B. EDUCE: a marriage of convenience: Prolog and a Relational DBMS. In Proc. Symp. in Logic Programming, 1986, pp. 36–45.
5. Ceri S., Gottlob G., and Tanca L. Logic Programming and Databases. Springer, Berlin, 1990.
6. Chakravarthy U.S., Minker J., and Grant J. Semantic query optimization: additional constraints and control strategies. In Proc. Expert Database Conference, 1986, pp. 345–379.
7. Chandra A.K. and Harel D. Horn clauses queries and generalizations. J. Log. Program., 2(1):1–15, 1985.
8. Comai S. and Tanca L. Termination and confluence by rule prioritization. IEEE Trans. Knowl. Data Eng., 15(2):257–270, 2003.
9. Kowalski R.A. and Kuehner D. Linear resolution with selection function. Artif. Intell., 2:227–260, 1971.
10. Laenens E. and Vermeir D. Assumption-free semantics for ordered logic programs: on the relationship between well-founded and stable partial models. J. Log. Comput., 2(2):133–172, 1992.
11. Lloyd J.W. Foundations of Logic Programming, 2nd edn. Springer, Berlin, 1987, ISBN 3-540-18199-7.
12. Sacca' D. and Zaniolo C. Stable models and non-determinism in logic programs with negation. In Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1990, pp. 205–217.
13. Tsur S. and Zaniolo C. LDL: a logic-based data language. In Proc. 12th Int. Conf. on Very Large Data Bases, 1986, pp. 33–41.
14. Ullman J.D. Principles of Database and Knowledge-Base Systems, Computer Science Press, Rockville, MD, USA, 1988.
15. W3C Member Submission. SWRL: a semantic Web rule language combining OWL and RuleML. 21 May 2004, Available at: http://www.w3.org/Submission/SWRL/.

Query Processor

Anastasia Ailamaki, Ippokratis Pandis
EPFL, Lausanne, Switzerland
Carnegie Mellon University, Pittsburgh, PA, USA

Synonyms

Query execution engine; Relational query processor; Query engine

Definition

The query processor in a database management system receives as input a query request in the form of SQL text, parses it, generates an execution plan, and completes the processing by executing the plan and returning the results to the client.

Key Points

In a relational database system the query processor is the module responsible for executing database queries. The query processor receives as input queries in the form of SQL text, parses and optimizes them, and completes their execution by employing specific data access methods and database operator implementations. The query processor communicates with the storage engine, which reads and writes data from the disk, manages records, controls concurrency, and maintains log files.

Typically, a query processor consists of four sub-components; each of them corresponds to a different stage in the lifecycle of a query. The sub-components are the query parser, the query rewriter, the query optimizer and the query executor [3].

The parser initially reads the SQL text, renames the table references to the *schema.table* template, and validates the structure. As a second step, it uses the database catalog to check the existence of the referenced tables, as well as ensuring that the user who submitted the specific query has the appropriate privileges for the particular operation on the particular data. If everything succeeds, the output from the parser is a data structure understood internally by both the rewriter and the optimizer. This data structure is handed over to the query rewriter.

The query rewriter modifies the query without changing its semantics. The rewriter replaces references to views as references to base tables, simplifies arithmetic expressions, and applies logical transformations to predicates. After the query is parsed and rewritten (and before it is passed on to the optimizer), the system checks a cache of execution plans of recently optimized queries for an execution plan for the specific query in order to avoid the (usually expensive) optimization phase.

The optimizer generates an efficient execution plan for answering a specific query. The decision on which specific access method or database operator implementation will be used relies heavily on the statistics kept by the system and the selectivity estimation. The output of the optimizer is the query execution plan. In the common case, the query plan is an interpretable dataflow directed acyclic graph, where each node is a specific implementation of a database operation. There are some systems where the optimizer generates directly executable machine code, such as in Daytona [2]. The optimizer can choose from many techniques that can speed up the execution of a query. For example, it may decide to generate a plan where multiple threads or processes work in parallel to answer a specific query. Such an execution strategy works well especially if the machine where the system is running contains multiple processors. Query optimization is the responsibility of a fairly sophisticated software module [1,3,4].

Although the query plan describes in detail the various operations needed for the execution of the query, it is the query executor that contains the algorithms for accessing base tables and indexes, as well as various database operator execution algorithms. The objective of the query executor is to execute the plan as fast as possible and return the answer to the client. The query optimizer and query executor are tightly coupled together. The query executor determines which algorithms implement the plans generated by the optimizer. For instance, if a query executor supports only Hash and Sort-Merge joins, then the optimizer is restricted to producing plans that use only those two join implementations.

Typically, query executors employ the *iterator* model, a simple and intuitive way to filter data. Each operator is implemented as a subclass of the iterator class, using as interface functions such as `init()`, `get_next()`, and `close()`. An iterator can be used as input to any other iterator, thereby enabling universal handling of iterators or iterator combinations by the system, regardless of the particular function they implement. However, recently researchers argue that the overhead of processing data in a tuple-at-a-time, iterator-based fashion leads to inefficient execution of queries when running on modern hardware and deep memory hierarchies [5].

There are many interpretations of what constitutes a query processor. The query executor is the subcomponent that does the “real” job of answering a query. It pulls the data out of the database and employs the various data manipulation algorithms towards them. Thus, a frequent misinterpretation is to consider the query processor and query executor as being synonyms.

Cross-references

- ▶ Hash Join
- ▶ Iterator
- ▶ Parallel Query Processing
- ▶ Query Optimization
- ▶ Query Plan
- ▶ Query Rewriting
- ▶ Sort-Merge Join

Recommended Reading

1. Graefe G. The cascades framework for query optimization. Q. Bull. IEEE TC on Data Engineering, 18(3):19–29, 1995.
2. Greer R. Daytona and the fourth-generation language Cymbal. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1999, pp. 525–526.
3. Hellerstein J.M., Stonebraker M., and Hamilton J. Architecture of a database system. Foundations and Trends in Databases, 1(2):141–259, 2007.
4. Selinger P.G., Astrahan M., Chamberlin D., Lorie R., and Price T. Access path selection in a relational database management system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 23–34.
5. Zukowski M., Boncz P., Nes N., and Heman S. MonetDB/X100 – a DBMS in the CPU cache. Q. Bull. IEEE TC on Data Engineering, 28(2):17–22, 2005.

Query Reformulation

▶ Web Search Query Rewriting

Query Rewriting

Evaggelia Pitoura
University of Ioannina, Ioannina, Greece

Synonyms

Query transformations

Definition

Query rewriting is one of the initial phases of query processing. After the original query is parsed and translated into an internal representation, query rewrite transforms it to an equivalent one by carrying out a number of optimizations that are independent of the physical state of the system. Typical transformations include un-nesting of subqueries, views expansions, elimination of redundant joins and predicates and various other simplifications.

Key Points

Query rewriting is one of the phases of query processing. It refers to the application of a number of transformations to the original query in order to produce an equivalent optimized one. Such transformations do not depend on the physical state of the system (such as the size of the relations, the system workload, etc). They are usually based on well-defined rules that specify how to transform a query expression into a logically equivalent one.

The goal of query rewriting is threefold: (i) the construction of a standardized starting point for query optimization (standardization), (ii) the elimination of redundancy (simplification), and (iii) the construction of expressions that are improved with respect to evaluation performance (amelioration).

To satisfy this goal, common responsibilities of the query rewriter include:

- View expansion
- Logical rewriting of predicates. For example, improving the match between expressions and the capabilities of index-based access methods
- Various semantic optimizations such as elimination of redundant joins and predicates
- Sub-query flattening

Typically, query rewriting is performed after parsing the original query. It can be thought of as either being the first part of query optimization or as an independent component preceding the query optimizer and the generation of the alternative execution plans. Rewriting is particularly important for complex queries, including queries with many sub-queries or many joins.

Cross-references

- ▶ Query Optimization
- ▶ Query Processing

Recommended Reading

1. Hellerstein J.M., Stonebraker M., and Hamilton J. Architecture of a database system. Foundations and Trends Databases, 1(2):141–259, 2007.
2. Jarke M. and Koch J. Query optimization in database systems. ACM Comput. Surv., 16(2):111–152, 1984.
3. Pirahesh H., Hellerstein J.M., and Hasan W. Extensible/rule based query rewrite optimization in starburst. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1992, pp. 39–48.

Query Rewriting Using Views

► Answering Queries Using Views

Query Suggestion

► Web Search Query Rewriting

Query Transformations

► Query Rewriting

Query Translation

ZHEN ZHANG
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Query translation; Query mapping

Definition

Given a source query Q_s over a source schema and a target query template over a target schema, query translation generates a query that is *semantically closest* to the source query and *syntactically valid* to the target schema. The semantically closest is measured by a closeness metrics, typically defined by precision and/or recall of a translated query Versus a source query over a database content. Syntax validness indicates the answerability of a translated query over the target schema. Therefore, the goal of query translation is to find a query that is answerable over the target schema and meanwhile retrieves the closest set of results as the source query would retrieve over a database content.

Historical Background

Query translation is an essential problem in any data integration system and has been studied extensively in the database area. Since a data integration system needs to integrate many different sources, query translation is thus needed to mediate heterogeneous query capabilities presented by those sources. A source typically only accepts and processes queries of certain formats. Such restrictions on acceptable queries form the query capability of the source. For instance, a Web database may only accept queries through their Web query interfaces, a relational database may accept SQL queries, and a legacy system may only accept selection queries over certain attributes through their wrappers.

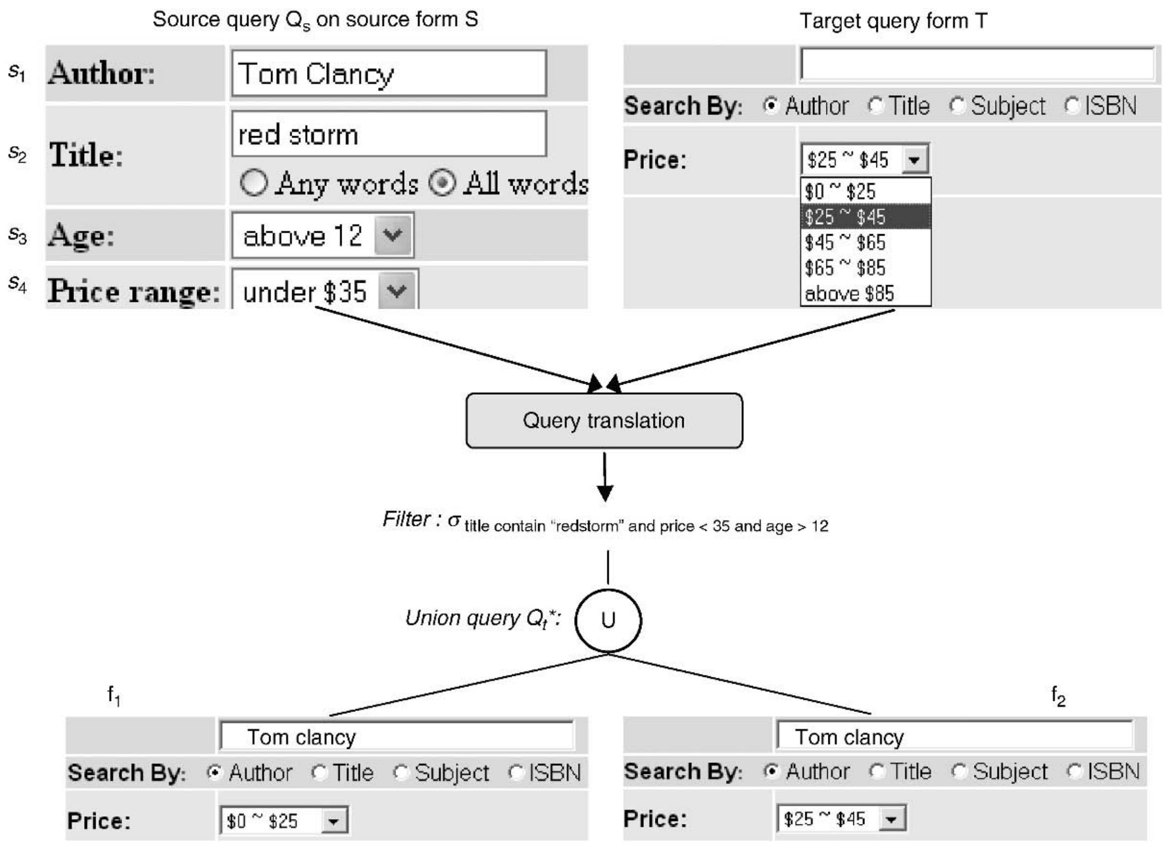
Foundations

To represent query capabilities, different description languages have been proposed. Vassalos [13] proposes p-Datalog, a datalog-like language for describing query capabilities. Halevy [8] uses capability records to describe, for accepted queries, their binding requirements of input parameters and the attribute name of output parameters. Rajaraman [12] proposes query templates in the format of parameterized queries to specify binding patterns of acceptable queries. Similarly, Zhang [15] uses predicate templates and form templates to represent query capabilities of Web databases through Web query interfaces. As an example, Fig. 1 shows two query forms with different query capabilities. The source query form S accepts conjunctive queries over four predicates: such as s_1 : [`author`; *contain*; Tom Clancy], s_2 : [`title`; *contain*; red storm], s_3 : [`age`; $>$; 12], and s_4 : [`price`; $\leq$; 35], i.e., $Q_s = s_1 \wedge s_2 \wedge s_3 \wedge s_4$. A target query form T supports predicate templates on `author`, `title`, `subject`, `ISBN` one at a time with an optional template on predicate price.

More specifically, the heterogeneity of query capabilities can be categorized into three levels:

Attribute Heterogeneity

Two sources may query a same concept using different attribute names. For instance, the source schema S in Fig. 1 supports querying the concept of reader's age, while the target schema T does not. Also, S denotes book price using price range, while T using price.



Query Translation. Figure 1. Form assistant: A translation example.

Predicate Heterogeneity

Two sources may use different predicates for the same concept. For instance, the price predicate in T has a different set of value ranges from those of S. As a result, a translated target predicate can only be as "close" to the source predicate as possible. Therefore, a closeness metrics needs to be introduced to set up a goal of translation. For instance, a *minimal subsumption* translation requires that a translated target query subsume the source query with fewest extra answers.

Query Structure Heterogeneity

Two sources may support different sets of valid combinations of predicates. In the above example, the target schema T only supports queries on one of the four attributes `author`, `title`, `subject` and `ISBN` at a time with an optional attribute `price`. Therefore, T cannot query author and `title` together, while S can.

The goal of query translation is to generate an appropriate query expressed upon the target schema T. Such a query, as Fig. 1 shows, in general, consists of two parts: a *union query* Q_t^* which is a union of queries upon the target schema to retrieve relevant answers from a target database, and a *filter* σ which is a selection condition to filter out false positives retrieved by Q_t^*. To minimize the cost of post processing, i.e., filtering, translation aims at finding a union query Q_t^* that is as "close" to the source query Q_s as possible so that it retrieves fewest extra answers. Q_t^* in Fig. 1 is such a query.

To realize the translation, query translation needs to reconcile the heterogeneities at the three levels – attribute, predicate and query. Techniques have been studied extensively for addressing the heterogeneity at each level.

Schema Matching for Attribute Heterogeneity

Schema matching (e.g., see the survey of [11]) focuses on mediating the heterogeneity at the attribute level. Recent schema matching approaches follow two different forms – pairwise matching and holistic matching. The pairwise matching approaches (e.g., [3,7]) take two schemas as input, and find best attribute matchings between the two. The holistic matching approaches, pursued by, e.g., [5,6,14], take a collection of schemas as input and generate a set of matchings over all these schemas. Different approaches suit different application settings. The holistic matching approaches are suitable for applications that would dynamically query a large scale of sources simultaneously, while the pairwise matching approaches are suitable for applications that query a small set of pre-configured sources.

Predicate Mapping for Predicate Heterogeneity

Predicate mapping focuses on addressing the heterogeneity at the predicate level. Existing solutions can be categorized into two categories: static predicate mapping mechanism and dynamic predicate mapping mechanism. The static predicate mapping mechanism works with a set of pre-configured data sources. It predefines mapping knowledge between a source schema and a target schema. In such scenarios, it is common, e.g., as [1] studies, to use pairwise rules to specify the mapping. Figure 2 gives some example rules that encode the mapping knowledge required for translation in the example of Fig. 1.

In contrast, the dynamic predicate mapping mechanism works with dynamically discovered sources in a domain. It does not encode the mapping knowledge for specific sources, but instead defines common domain-based translation knowledge that handles most sources in a domain. Such a system may use rules to encode domain knowledge or alternatively may use a search-driven mechanism to dynamically search for the best mapping. Such a search-driven mechanism "materializes" the semantics of a query as results over a database. For instance, to realize rule r_3 in Fig. 2 in the search mechanism, the dynamic mechanism projects both the source and target predicates onto an axis of real numbers, and thus compares their semantics based on their coverage. Finding the closest mapping thus naturally becomes a search problem – to search for the ranges expressible in the target form that minimally cover the source predicate.

Query Rewriting for Query Structure Heterogeneity

Capability-based query rewriting focuses on mediating the heterogeneity at query structure level. Most query rewriting works [4,8,9,10,12] are studied for data integration systems following a *mediator-wrapper* architecture, where a global mediator integrates local data sources through their wrappers. Query rewriting specifically studies the problem of how to mediate a "global" query (from the mediator) into "local" subqueries (for individual sources) based on their query capabilities. There are two basic approaches for addressing query rewriting in data integration system – *global as view* (GAV) and *local as view* (LAV). In global as view, each relation in the mediated (global) schema is defined as a view over schemas of local data source. Query rewriting in GAV is straightforward – simply replacing relation names in a query (over global schema) with their view definitions will yield a valid rewriting of query (over local schemas). In contrast, in local as view, each relation of a local schema is defined as a view over the mediated global schema. Query rewriting in LAV is thus to find a query plan or query expression which uses only views (i.e., local schemas) to answer queries over global schema. In particular, this problem is often abstracted as *answering queries using views*. For a thorough survey of related techniques for answering query using views, please refer to [4].

Key Applications

Query translation is a key component in any data integration system. The broad range of applications for data integration gives rise to the diverse applications of query translation. Two examples are:

Vertical Integration Systems

A vertical integration system integrates information from multiple pre-configured sources (usually in the same domain of data), and thus requires translating queries from a unified query interface to individual data sources. As data sources are usually pre-configured, such a system usually replies on static query translation mechanism such as [1] to handle translation with predefined source knowledge.

r_1	[author; *contain*; \$t] → *emit:* [author; *contain*; \$t]
r_2	[title; *contain*; \$t] → *emit:* [title; *contain*; \$t]
r_3	[price; *under*; \$t] → **if** \$t ≤ 25, *emit:* [price; *between*; 0,25]
	elif \$t ≤ 45, *emit:* [price; *between*; 0,25] ∨ [price; *between*; 25,45]
	……

Query Translation. Figure 2. Example mapping rules of source *S* and target *T*.

Meta Querying Systems

A meta querying system, e.g., [2], integrates dynamically selected sources relevant to user's queries, and on-the-fly translates user's queries to these sources. As sources are dynamically discovered without predefined source knowledge, such a system needs a dynamic query translation mechanism such as [15] which handles translation without relying on source-specific knowledge.

Cross-references

▶ Information Integration
▶ Query Rewriting
▶ Query Rewriting Using Views
▶ Schema Matching
▶ View-Based Data Integration

Recommended Reading

1. Chen-Chuan C.K. and Garcia-Molina H. Approximate query mapping: Accounting for translation closeness. VLDB J., 10(2–3):155–181, September 2001.
2. Chen-Chuan C.K., He B., and Zhang Z. Toward large scale integration: Building a metaquerier over databases on the web. In Proc. 2nd Biennial Conf. on Innovative Data Systems Research, 2005, pp. 44–55.
3. Doan A., Domingos P., and Halevy A.Y. Reconciling schemas of disparate data sources: A machine-learning approach. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001, pp. 509–520.
4. Halevy A.Y. Answering queries using views: A survey. VLDB J., 10(4):270–294, 2001.
5. He B. and Cheng-Chuan C.K. Statistical schema matching across web query interfaces. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 217–228.
6. He B., Cheng-Chuan C.K., and Han J. Discovering complex matchings across web query interfaces: A correlation mining approach. In Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2004, pp. 148–157.
7. Kang J. and Naughton J.F. On schema matching with opaque column names and data values. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 205–216.
8. Levy A.Y., Rajaraman A., and Ordille J.J. Querying heterogeneous information sources using source descriptions. In Proc. 22th Int. Conf. on Very Large Data Bases, 1996, pp. 251–262.
9. Papakonstantinou Y., Gupta A., Garcia-Molina H., and Ullman J.D. A query translation scheme for rapid implementation of wrappers. In Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases, 1995, pp. 161–186.
10. Papakonstantinou Y., Gupta A., and Haas L. Capabilities-based query rewriting in mediator systems. In Proc. Int. Conf. Parallel and Distributed Information Systems, 1996, pp. 170–181.
11. Rahm R. and Bernstein P.A. A survey of approaches to automatic schema matching. VLDB J., 10(4):334–350, 2001.
12. Rajaraman A., Sagiv Y., and Ullman J.D. Answering queries using templates with binding patterns. In Proc. 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1995, pp. 105–112.
13. Vassalos V. and Papakonstantinou Y. Expressive capabilities description languages and query rewriting algorithms. J. Logic Program., 43(1):75–122, 2000.
14. Wu W., Yu C.T., Doan A., and Meng W. An interactive clustering-based approach to integrating source query interfaces on the deep web. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 95–106.
15. Zhang Z., He B., and Chen-Chuan Chang K. Light-weight domain-based form assistant: querying web databases on the fly. In Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 97–108.

Query Tree

▶ Query Plan

Query Tuning

▶ Application-Level Tuning

Querying DNA Sequences

▶ Query Languages and Evaluation Techniques for Biological Sequence Data

Querying Protein Sequences

▶ Query Languages and Evaluation Techniques for Biological Sequence Data

Querying Semi-Structured Data

▶ Structured Document Retrieval

Queuing Analysis

▶ Performance Analysis of Transaction Processing Systems

Queuing Mechanism

▶ Scheduler

Queuing Systems

▶ Message Queuing Systems

Quorum Systems

Marta Patiño-Martinez
Universidad Politecnico de Madrid, Madrid, Spain

Definition

Replication is a technique that provide scalability and high availability by introducing redundancy. This entry focuses on full replication for simplicity, that is, each node (site) has a copy of the entire database. Therefore, the terms nodes and replicas are used interchangeably. Replication increases performance (scale-out) as access to the data can be distributed across the replicas. Furthermore, the data remains available as long as some replicas are accessible. The most common approach is to execute all write operations (updates) at all replicas in order to keep them consistent while read operations are executed at a single replica. For read-intensive workloads this achieves the desired scalability. However, this approach, known as read-one/write-all (ROWA), has poor availability for write operations (writes cannot operate once a single replica fails) and does not provide scalability under write-intensive workloads. Quorum systems address both of these issues. They reduce the number of copies involved in write operations at the cost of increasing the number of copies involved in read operations. Reducing the number of copies to be accessed implies increased availability, tolerance of network partitions, reduced communication costs and the possibility of balancing the load among replicas.

In order to preserve data consistency of write operations, and a read and a write operation on the same data item must overlap at least at one replica. For that, each operation has to execute on a subset of replicas called a quorum. A *quorum system* over a set of nodes $\mathcal{N} = \{N_1,...N_n\}$ is defined as a collection $\mathcal{S}$ of subsets $S_i \subseteq \mathcal{N}$ with pair-wise non-null intersection. This means, for each $S_i, S_j \in \mathcal{S}$, $S_i \cap S_j \neq \emptyset$. Each subset $S_i \subseteq \mathcal{N}$ is called a *write quorum.* A write operation on a data item of the database must be executed in a write quorum. The requester asks all sites in the quorum for permission. If all of them grant permission, the write operation is executed at all nodes in the quorum. Data items are tagged with versions. If a write operation succeeds in a write quorum S_i, all nodes in S_i set the version of the affected data item to the same value, namely a value higher than any current version among the nodes in the quorum. The non-empty intersection property guarantees that only one quorum can make a decision at a time. That is, if two concurrent write operations on the same data item ask for permission to two write quorums, at least one node is member of both write quorums and gives permission to only one of the write operations. Read operations are executed on a *read quorum*, a set of sites $R_j \in \mathcal{N}$ such that $\forall S_i \in \mathcal{S}, S_i \cap R_j \neq \emptyset$. The requester of the read operation executes the read at all the sites of the read quorum, which return the value read and the version. The requester selects the value corresponding to the highest version. Since a read quorum intersects with all write quorums, it is guaranteed that at least one of the versions read is the latest one.

Historical Background

Quorum systems for data replication were proposed concurrently by [13] and [6] in order to provide availability despite individual node failures and network partitions. *Majority* quorum (also known as *quorum consensus*) [13] exploits the concept of majority to guarantee the intersection property. A quorum can be any majority of nodes. *Weighted voting* [6] generalizes majority by assigning votes to each site and defining a quorum to be a majority over the total number of votes in the system. An overview of early quorum systems is given in [4].

Maekawa [9] initiated a research line for increasing the scalability of quorum systems by reducing their size (in the order of $O(\sqrt{n})$). After this seminal work a large number of quorum systems exploiting different schemes have been proposed, mainly in the nineties. Extensive surveys on quorum systems can be found in [11,7].

Many of the proposed quorum systems exploit geometrical properties to satisfy the intersection property. *Grid quorums* arrange sites as a grid and then define read and write quorums as rows and columns to enforce their intersection. Grids quorums can be rectangular [5] or can have other shapes, such as triangles [11]. Grid quorums were generalized into hierarchical grid quorums by [8]. Another popular way to arrange sites are trees. Tree quorums were introduced by [1].

Quorums have been extended to tolerate Byzantine (arbitrary) failures requiring the intersection of two quorums to be bigger than one. Such quorums have been termed Byzantine quorums [10].

The properties of quorums have been studied extensively, especially availability and scalability. Optimal availability for sites with homogeneous failure probability were studied in [2,3,12]. Scalability (also known as load) has been first studied for symmetric update processing (all the sites in the write quorum fully execute the update transaction or operation) in [11], and then under asymmetric update processing (only one site executes the operations, the others only apply the changes) in [7].

Foundations

There are two basic ways to define quorum systems, plain (or exclusive) quorums and read/write quorums. Read/write quorums allow reduction of cost of read operations by exploiting the knowledge of whether an operation is a read or a write. They are more adequate for data replication where this knowledge is typically exploited. Plain quorums, or just quorums, are typically defined for mutual exclusion purposes, but can also be used for data replication by using exclusive quorums for both reads and writes.

A set system $\mathcal{S}$ is a collection of subsets $S_i \subseteq \mathcal{N}$ of a finite universe $\mathcal{N}$. A quorum system defined over a set of sites $\mathcal{N}$ is a set system $\mathcal{S}$ that fulfils the following property: $\forall S_i, S_j \in \mathcal{S},\ S_i \cap S_j \neq \emptyset$. Given a quorum system $\mathcal{S}$, each $S_i \in \mathcal{S}$ is a quorum. A read-write quorum system, over the set of sites $\mathcal{N}$, is a pair (R,W) where W is a quorum system (write quorums), and R a set system (read quorums) with the following property: $\forall\ W_i \in W,\ \forall\ R_j \in R,\ W_i \cap R_j \neq \emptyset$.

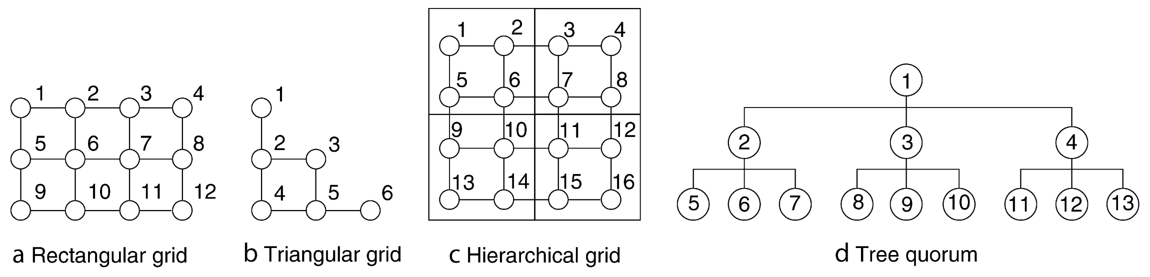
Quorum Types and Their Sizes

For scalability purposes it is beneficial to keep both read and write quorums as small as possible. Furthermore, in order to distribute the load fairly, each node should ideally participate in the same number of quorums. In the following, n denotes the number of nodes in $\mathcal{N}$.

Majority In the majority quorum system [13] read and write quorums must fulfill the following constraints (wq and rq stand for the write and read quorum sizes): $2 \cdot wq > n$ and $rq + wq > n$. The minimum quorum sizes satisfying these constraints are: $2 \cdot wq = n+1$ and $rq + wq = n+1$ and therefore, $wq = \lfloor \frac{n}{2} \rfloor + 1$ and $rq = \lceil \frac{n}{2} \rceil = \lfloor \frac{n+1}{2} \rfloor$. The ROWA approach can be seen as an extreme case of majority, in which $rq=1$ (a single replica is read) and $wq = n$ (all replicas are written). An example of a majority quorum system for three sites (1,2,3) with $rq = wq = 2$ is: {{1, 2}, {2, 3}, {1, 3}}. The majority quorum system is fair since each node has the same probability to be part of a quorum. In weighted majority [6], each site has a non-negative weight (votes). A write quorum consists of nodes such that the sum of their votes is more than half of the total number of votes. A read quorum consists of nodes that have at least half of the total number of votes. Assigning votes allows to adjust to heterogeneous environments where nodes have different processing power and availabilities.

Grids Another family of quorums is grid quorums. The simplest form of grid quorum is the rectangular grid [5]. A rectangular grid quorum organizes n sites in a grid of r rows and c columns (i.e., $n = r \cdot c$). Figure 1a depicts a 3×4 rectangular grid. A read quorum consists of accessing an element of each column of the grid ($rq = c$). A write quorum consists of a full column and one element from each of the remaining columns ($wq = r + c - 1$). In the quorum system of Fig. 1a {1,10,7,12} would be a read quorum and {2,6,10,5,3,8} a write quorum. The rectangular grid with the optimal (smallest) quorum size is the square. In this case, $rq = \sqrt{n}$ and $wq = 2 \cdot \sqrt{n} - 1$.

A variation of rectangular grids are *hierarchical grids* [8]. For instance, 16 sites can be configured into a two-level-grid with 2×2 grids at each level (Fig. 1c).



Quorum Systems. Figure 1. Different quorum systems.

A hierarchical grid organizes sites into a multi-level hierarchy, such that they reside on the leaves of this hierarchy, while other levels are represented by logical nodes. Each node at level i of the hierarchy (beside leaves) is defined by a rectangular $m \times n$ grid of nodes at level $i+1$. Two constructions are used to build quorums: row covers and full rows. A row cover is formed recursively by selecting a set of $(i+1)$-level nodes where each node pertains to a different row of the grid. A full row is formed recursively by selecting at level i a set of $(i+1)$-level nodes all pertaining to a single row of the grid. A read quorum consists of a row cover. A write quorum consists of the union of a full row and a row cover. A read quorum would be $\{1,6,7,8\}$ and a write quorum would be $\{1,5,6,7,8, 10,14\}$. For square grids, this results in a read quorum size of $\sqrt{n}$ and a write quorum size of $2 \cdot \sqrt{n} - 1$, i.e., identical to its non-hierarchical counterpart.

Another way to arrange a grid quorum is a *triangular grid* [11]. Sites in a triangular grid are arranged in d rows such that row i $(1 \leq i \leq d)$ has i elements (Fig. 1b). A write quorum is defined as the union of one complete row and one element from every row below the full row. Therefore, the quorum size is always d. Read quorums are either a write quorum or an element from each row. In Fig. 1b a sample quorum would be $\{2,3,5\}$.

Triangle quorums are not fair since nodes that are higher in the triangle are more likely to be part of a quorum. In contrast, both rectangular and hierarchical grid quorums are fair.

Trees Another important family of quorum systems is tree quorums. Tree quorums were introduced in [1]. Similar to grid quorums, tree quorums are arranged as a logical structure over the nodes in order to reduce quorum sizes. The nodes are organized into a tree of height h and degree d, i.e., each inner node in the tree has d children. Figure 1d shows a tree with $d{=}h{=}3$. A *tree quorum* $q = \langle l, b \rangle$ over a tree with height h and degree b is a tree of height l and degree b constructed as follows. Read and write quorums have the same structure. The quorum contains the root of the tree and b children of the root. Then, recursively for each selected child, b of its children have to be selected, and so on, until a depth l is reached. In the case all nodes are accessible the quorum forms a tree of height l and degree b. If some node is inaccessible at depth h' from the root, the node is replaced by b tree quorums of height $l - h'$ starting from the children of the inaccessible node. In order to guarantee intersecting quorums, quorums must overlap both in height and degree. A read quorum $rq = \langle l_r, b_r \rangle$ and write quorum $wq = \langle l_w, b_w \rangle$ overlap if $l_r + l_w > h$ and $b_r + b_w > d$. Two write quorums overlap if $2 \cdot l_w > h$ and $2 \cdot b_w > d$. Tree quorums are generally not fair since nodes that are closer to the root take part in more quorums.

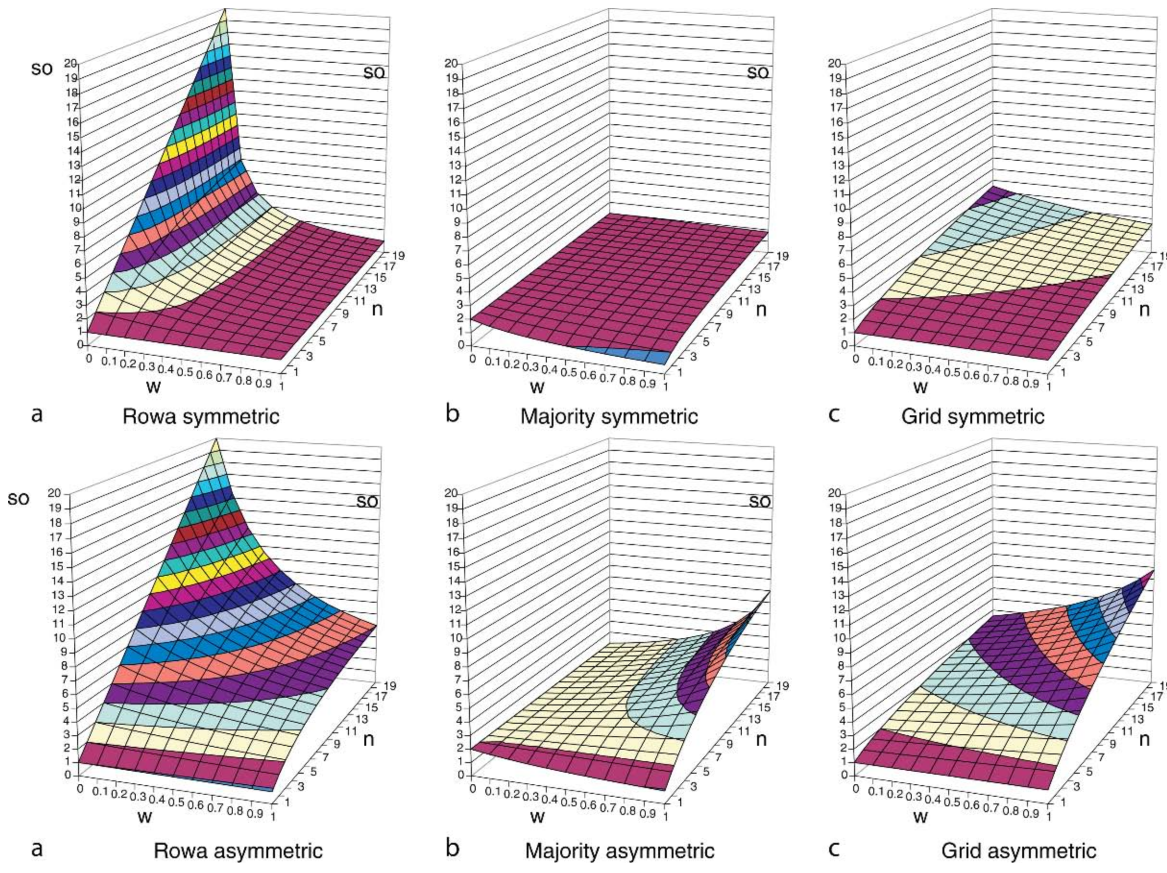
Depending on the values of l and b, different tree quorum systems can be defined. The most well-known is the *ReadRoot* in which a write quorum $wq = \langle h, d/2 + 1 \rangle$, i.e., at each level in the tree a majority of nodes needs to be accessed while a read quorum is $rq = \langle 1, (d+1)/2 \rangle$. That is, all reads go to the root. If the root fails, reads go to the next level while writes cannot be performed anymore. More availability is provided by *MajorityTree* where a majority approach is used both for the degree and height parameters. That is, a read quorum is $rq = \langle (h+1)/2, (d+1)/2 \rangle$ and a write quorum is $wq = \langle h/2 + 1, d/2 + 1 \rangle$. This increases the availability of write operations (since write quorums can be built without the root) but also the access costs of read operations. For the tree depicted in Fig. 1d, {1, 2, 3} is a read and a write quorum, or, if the root is down, {2, 5, 6, 3, 8, 9}.

Availability

Availability of quorums has also been studied extensively. Barbara and Garcia-Molina [3] demonstrated that majority is the most available quorum system for homogeneous failure probabilities (all sites have the same failure probability), if the failure probability p is higher than 0.5. Later, it was shown in [12] that for $p<0.5$ monarchy (all access goes to a single node) was the most available quorum system. When the failure probabilities are heterogeneous, the most available quorum system is weighted majority. The computation of the optimal weights has been provided for the different cases: all sites with $p>0.5$ [14] and the general case in which $0<p<1$ [2]. An extensive comparative study of the availability of different quorum systems is presented in [12].

Experimental Results

A comparison of the performance of quorum systems has been performed by [7,11]. In [7], two different forms of update processing were considered. Using symmetric update processing all the nodes in the write quorum fully execute update transactions (i.e., update operations). In contrast, asymmetric processing lets one node execute the operation while the others only apply the changes. As applying changes typically requires less resources than fully executing the operation, asymmetric processing imposes less load per write operation on the system than symmetric processing. Figure 2 compares the scalability of ROWA, majority and rectangular grids using the analytical model from [7]. The x-axis shows the the fraction of writes ($w = 1.0$ means 100% write operations, $w = 0.0$ means 100% reads). The y-axis shows the total number of nodes (replicas). The z-axis shows the scalability (how many times the throughput of a single-node system is multiplied by the replicated system). The first row of figures shows the performance for symmetric update processing. The scalability is generally very poor, especially for majority and grid. The second row shows results for asymmetric update processing when applying writes has 15% of the costs of fully executing the operation. Scalability



Quorum Systems. Figure 2. Scalability of majority.

improves substantially. Majority and grid quorums can improve over ROWA for write-intensive workloads, but at the cost of performing worse in read-intensive environments.

Key Applications

One of the main applications of quorum systems is data replication. However, they are also used for other decentralized control protocols such as distributed mutual exclusion, distributed consensus, Byzantine replication, and group membership.

Future Directions

One of the main open issues with data replication based on quorum systems is how to manage efficiently collections of objects such as tables. Quorum systems might work reasonably well for accesses to individual objects. However, the access of collections of objects has not yet been adequately addressed. When accessing collections of objects the system is forced to collect all the instances of the collection from a read quorum to obtain the latest version of every object and only then, it becomes possible to select the subset of objects from the collection (typically by means of a predicate as in a SELECT statement). This compilation of the full collection from all the sites in the quorum at a single site ruins the performance and scalability of the quorum approach.

Recommended Reading

1. Agrawal D. and Abbadi A.E. The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. ACM Trans. Database Syst., 17(4):689–717, 1992.
2. Amir Y. and Wool A. Optimal Availability Quorums Systems: Theory and Practice. Inf. Proc. Letters, 65(5):223–228, 1998.
3. Barbara D. and Garcia-Molina H. The reliability of vote mechanisms. IEEE. Trans. Comput., 36:1197–1208, 1987.
4. Bernstein P.A., Hadzilacos V., and Goodman N. Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987.
5. Cheung S.Y., Ahamad M., and Ammar M.H. The grid protocol: a high performance scheme for maintaining replicated data. In Proc. 6th Int. Conf. on Data Engineering, 1990, pp. 438–445.
6. Gifford D.K. Weighted Voting for Replicated Data. In Proc. 7th ACM Symp. on Operating System Principles, 1979, pp. 150–162.
7. Jiménez-Peris R., Patiño-Martínez M., Alonso G., and Kemme B. Are Quorums an Alternative for Data Replication. ACM Trans. Database Syst., 28(3):257–294, 2003.
8. Kumar A. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. IEEE Trans. Comput., 40 (9):996–1004, 1991.
9. Maekawa M. A Algorithm for Mutual Exclusion in Decentralized Systems. ACM Trans. Computer Syst., 3(2):145–159, 1985.
10. Malkhi D., Reiter M.K., and Wool A. The Load and Availability of Byzantine Quorum Systems. SIAM J. Comput., 29(6): 1889–1906, 2000.
11. Naor M. and Wool A. The Load, Capacity, and Availability of Quorum Systems. SIAM J. Comput., 27(2):423–447, 1998.
12. Peleg D. and Wool A. The Availability of Quorum Systems. Information and Computation, 123(2):210–223, 1995.
13. Thomas R.H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM Trans. Database Syst., 4(9):180–209, 1979.
14. Tong Z. and Kain R.Y. Vote Assignments in Weighted Voting Mechanisms. In Proc. 7th Symp. on Reliable Distributed Syst., 1988, pp. 138–143.