Online Processing Redux

Joseph M. Hellerstein
U.C. Berkeley

Abstract

The term "online" has become an all-too-common addendum to database system names of the day. In this article we reexamine the notion of processing queries online. We distinguish between online processing and preprocessing, and argue that online processing for large queries requires redesigning major portions of a database system. We highlight pressing applications for truly online processing, and sketch ongoing research in these applications at Berkeley. We also outline basic techniques for running long queries online. We close by reevaluating the typical measurements of cost/performance for online systems, and propose a mass-market approach for designing and measuring data-intensive processing.

1 Introduction

In the parlance of today's database systems, "online" signifies "interactive", "within the bounds of patience." Online processing is the opposite of "batch" processing. In the dark days of computing, all serious work was done in batch mode — the COBOL or FORTRAN programmer submitted her "job" to the machine operator and busied herself with other tasks; the expectation was that the computer would be occupied for some time generating a solution. By contrast, newer "online" systems would return an answer while the programmer remained connected to the system. Because early systems typically required users to wait for the completion of one job before starting another, online processing had to have interactive performance.

For many of today's programmers – this author included – batch processing as described above is a distant memory. Yet there are still a variety of scenarios where computers tax the patience of both programmers and naive users. Most of these scenarios arise when processing significant amounts of data, either through a query engine, a network, or both. The prevalance of these problems in data-intensive systems has led many database vendors to describe their systems' processing as "online", particularly the recently much-ballyhooed "On Line Analytic Processing" (OLAP) systems.

In this paper, we revisit the phrase "online" as it applies to data-intensive applications. First, we argue that the term should be used more carefully — in particular, we point out that many of today's so-called OLAP systems do not process online at all. Second, we highlight a large class of applications which require online processing over large data sets. We present a suite of techniques for achieving online performance even for large, data-intensive jobs. We close by calling into question the commonly-held economic metrics for data-intensive systems.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 Today's OLAP? PAP

The OLAP products available today provide interactive drill-down, roll-up and cube facilities [GBLP96]. Implementation techniques differ: some systems implement these facilities over relational query engines ("ROLAP"), others implement special-purpose multidimensional storage and retrieval engines ("MOLAP"). The interfaces are indeed "online", in the sense that users see interactive performance — with each "point" or "click", information is consolidated and displayed almost instantaneously.

OLAP operations typically involve major fractions of large databases. It should therefore be surprising that today's OLAP systems provide interactive performance. The secret to their success is *precomputation* — in most OLAP systems, the answer to each point and click is computed long before the user even starts the application. In fact many OLAP systems do that computation relatively inefficiently [ZDN97], but since the processing is done in advance the end-user does not see the performance problem.

Observe that today's OLAP systems do essentially no processing — online or otherwise — while the end-user is running them. They are better termed Precomputed Analytic Processing (PAP) systems. These systems have inherent space and time problems. Everything a user can do at the front end must be at least partially precomputed in order to guarantee interactive performance. This means that the system must precompute and store a large amount of information, some of which may never be used. This approach remains untenable today for large databases.

1.2 Truly Online Processing

Today's OLAP systems were intended to be a workable alternative to the batch-like speeds of traditional query engines, which discourage large scale data "browsing" and analysis. If traditional engines have batch behavior, and today's OLAP systems are really PAP, how do we implement truly online processing for large-scale data-intensive problems?

The answer is to change query processing — and indeed all data-intensive software — so that it runs in an online fashion. This does not mean traditional performance improvements involving expensive equipment and hand-tuned software. Rather it involves:

1. Carefully redesigning algorithms so that long-running operations return steady, incrementally improving estimates.

2. Allowing users to control the behavior of long-running operations on the fly.

Developing software that behaves this way requires a significant shift in focus, and major changes in implementation. It cannot be done with a simple object-relational "plugin module" or web "applet". On the other hand, as we show below it is possible to provide online processing with clean, modular modifications to a database system or application.

In the remainder of this paper we highlight areas where truly online processing is becoming critical, and then describe some basic implementation techniques to make it possible.

2 Critical Applications for Online Processing

Batch-style behavior remains a problem in a number of applications. The result of this is either that the application is frustratingly slow (discouraging its use), or the user interface prevents the application from entering batch states (constraining its use.) The applications in this section are currently being handled with one or both of these approaches.

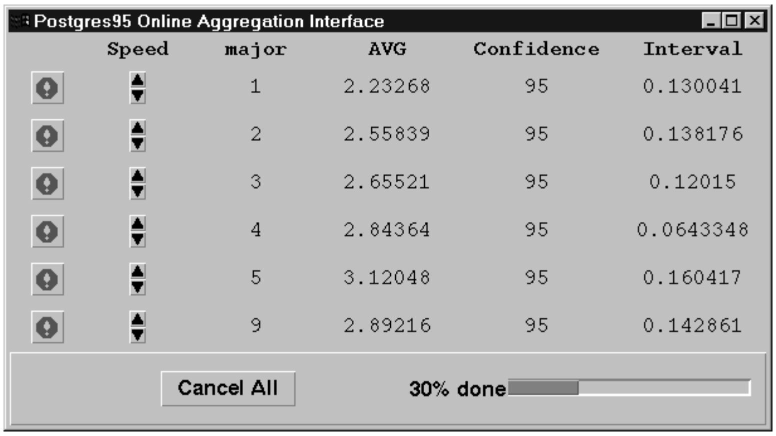


Figure 1: A Speed-Controllable Multi-Group Online Aggregation Interface

2.1 Online Aggregation

Aggregation queries in relational database systems often require scanning and analyzing a significant portion of a database. In current relational systems such queries have batch behavior, requiring a long wait for the user. In [HHW97] we detail an Online Aggregation system we are building at Berkeley, which performs relational aggregation queries in an online fashion.

Consider the following simple relational query:

```
SELECT AVG(final_grades) FROM grades
GROUP BY major;
```

The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in Figure 1. For each output group, the user is given a current estimate of the final answer, and a "confidence interval", which says that with $x\%$ confidence, the current estimate is within an interval of size k from the final answer. A status bar at the bottom of the screen shows how much processing time remains. These interfaces expose salient features of the current state of processing, and predict the final outcome in a statistically accurate fashion. In addition, controls are provided to stop processing on a group, or to speed up or slow down the group relative to others. These interfaces require the support of significant modifications to a relational DBMS. Many of the implementation themes which support online aggregation are described in Section 3.

2.2 Online Data Mining

Data mining algorithms typically run for hours on large datasets without producing output. While running, they usually make at least one complete pass over the database. A particularly frustrating aspect of data mining algorithms is that they are not only slow, but also opaque; they are "black boxes" that do not allow users to tune them in significant ways without starting over.

Consider, for example, the common algorithms for finding "association rules" in market-basket data [AS94]. The goal of these algorithms is to find sets of items that people tend to buy together in a single trip to a store (e.g., rules of the form "people who buy diapers also tend to buy beer").

To use an association rule application, a user specifies values for two variables, one that sets a minimum threshhold on the amount of evidence required for a set of items to be produced (*minsupport*) and another which sets a minimum threshhold on the correlation between the items in the set (*minconfidence*). The system begins its multi-hour undertaking, at the end of which it produces association rules which passed the threshholds of minimum support and confidence. Woe to the user who sets those threshholds incorrectly! Setting them too high means that few rules are returned and the process must be restarted. Setting them too low means that the system (a) runs even more slowly, and (b) returns an overwhelming amount of information, most of which is useless.

The association rules techniques can be easily "brought online", and a prototype implementation has been undertaken at Berkeley [ASY97]. While the algorithm is in its first phase producing individual items (1-itemsets) that have minimum support, it can provide running estimations of support for 1-itemsets in the same manner as an online aggregation query for `COUNT`. During this process, various items can be removed from consideration by the user, potentially speeding up computation of larger itemsets. Similarly, the user can modify minsupport and minconfidence as desired. Later phases of the algorithm are somewhat different from online aggregation processing, but still allow the user to add and remove itemsets from consideration, modify minsupport and minconfidence, and watch itemsets being generated. When a user changes the itemsets considered at any level (e.g., deletes a 2-itemset S from consideration), the system must spawn a new thread to "percolate" this change upwards through larger itemsets (e.g., delete all itemsets that are supersets of S). When all threads of the online data miner terminate, the user is still free to modify the support and confidence, or explicitly cause itemsets of any size to be added to or deleted from consideration — the system simply starts a new thread to differentially handle any newly added or deleted itemsets from that stage onward.

Most other data mining algorithms (clustering, classification, pattern-matching) are similarly time-consuming, and also relatively easily brought online. Note that interactive, online data mining is closer to data *browsing* — this should be particularly appealing to users who are skeptical that a computer can find all their answers automatically.

2.3 Online GUI Widgets

A common criticism of database systems is that they are much harder to use than desktop applications like spreadsheets. A common criticism of spreadsheets is that they do not gracefully handle large datasets. An inherent problem is that many spreadsheet behaviors are painfully slow on large datasets, despite impressive improvements from the commercial vendors. The problems typically have to do with the point-and-click nature of the GUI widgets used in spreadsheets.

Consider the common "list box" widget — it allows the user to bring up a list and scroll through it, or jump to particular points by typing prefixes of words in the list. Now imagine implementing a list box over gigabytes of unsorted, unindexed data resulting from an *ad hoc* query. This is likely to be rather unpleasant to use. Similarly unpleasant behavior arises when sorting, grouping, recalculating or cross-tabulating over large datasets — activites which are usually interactive in spreadsheets. Today's desktop applications carefully, almost imperceptibly discourage these behaviors over large unindexed datasets. There are many scenarios where such behavior is desirable, however; in such cases unwieldy database software (e.g., OLAP and Data Warehousing) is employed, requiring batch performance and lots of disk space to set up.

It should be possible to put online processing behind many of the widgets found in desktop applications so they continue to run smoothly over large datasets. Many development environments and toolkits today include libraries of GUI widgets. A data-intensive online GUI widget is a natural extension. In some instances, online GUI widgets could present different interfaces than standard widgets, in order to represent the state of ongoing processing — a trivial example is the "heartbeat" given by many web browsers as they present text and images online. In other instances status information might be unnecessary. The code underneath these widgets would need to process data using the kinds of techniques described in Section 3.

2.4 Online Data Visualization

Data visualization is an increasingly active research area, with some impressive prototypes under development (e.g. Tioga Datasplash [ACSW96], DEVise [LRB+97], Pad [PF93]). These systems are all interactive, allowing users to "pan" and "zoom" in datasets, and derive and view new visualizations quickly.

An inherent challenge in architecting a data visualization system is that it must present large volumes of information efficiently. This involves scanning, aggregating and rendering large datasets at point-and-click speeds. Typically these visualization systems do not draw a new screen until its image has been fully computed. Once again, this means batch-style performance for large datasets. This is particularly egregious for visualization systems that are expressly intended to support browsing of large datasets.

A natural solution is to extend a visualization system to draw objects as soon as they are fetched from the database. For example, as soon as a tuple is fetched, a corresponding point can be plotted on a map or graph. An more complex alternative we are exploring is to combine this incremental, sampling-like access with network data encoding. We model the final state of the screen as a *single aggregate object* to be estimated. After scanning a number of tuples, we use them as a sample to estimate the first few coefficients of a wavelet encoding of the final image. As more tuples are scanned, this estimate is refined. The user sees the picture improve much the way that images become refined during network transmission. This may be particularly useful when a user pans or zooms on a canvas, when the accuracy of what is seen is not as important as the rough outlines of the moving picture.

3 Basic Techniques for Online Processing

We believe that truly online processing can be built out of a number of relatively simple operators, much as traditional query processing engines are built. In this section we highlight some of the techniques we are studying. We make no claim to be exhaustive. However these techniques appear to be effective for applications we have studied, and are suggestively different from traditional algorithms for handling large datasets.

3.1 Sampling and Statistics

For an online estimation to be representative, it has to be built on some appropriate sample set of inputs. Olken proposed sampling access methods [Olk93], and these are quite applicable for applications which require guarantees of randomness during online processing. In many cases more traditional access methods may be used, as long as the order of access is expected to be uncorrelated with the estimation being made.

In the Online Aggregation project, we have exploited and developed results in statistics to provide estimators for common SQL aggregation functions (`COUNT, SUM, AVG, STDDEV`), and confidence intervals of the sort seen in Figure 1 [Haa96, Haa97a]. These kinds of estimations are obviously useful for Online Aggregation. They may also be useful in applications like online visualization and mining, where the system needs to decide when to modify previous estimations presented to users.

3.2 Striding Access Methods

Given a dataset which can be decomposed into groups, *striding* access methods provide delivery of tuples from the different groups at user-controllable relative rates. The Online Aggregation interface in Figure 1 bases its "speed" buttons on striding access methods.

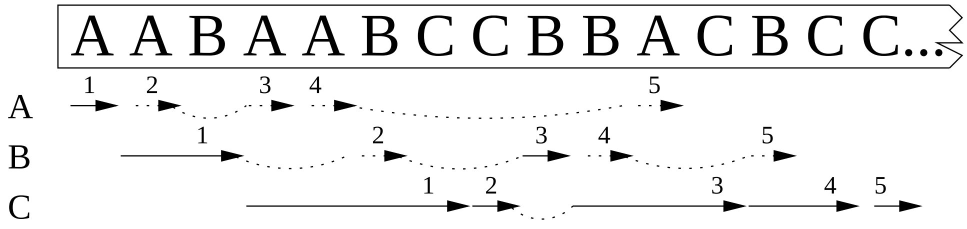


Figure 2: Heap Striding. The large letters represent the group attributes of tuples in a heap file. The three rows of arrows represent the accesses per group, with solid lines indicating initial I/Os and dotted lines indicating tuples which are revisited for delivery. The numbers represent the "strides" through the groups.

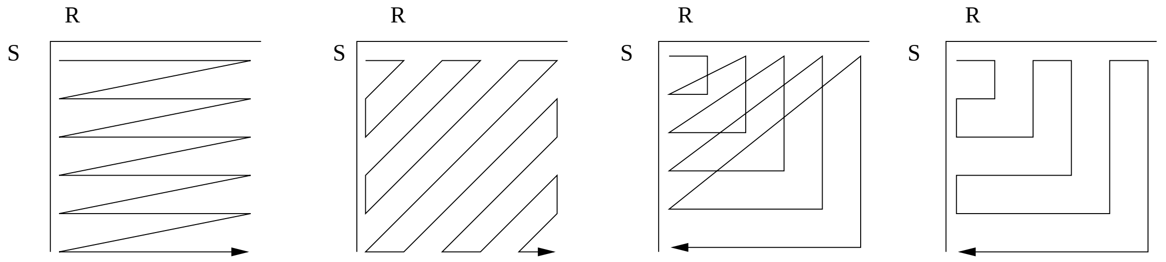
The Index Stride access method was presented in [HHW97]. Given a B-tree index on the grouping columns,[1] on the first request for a tuple we open a scan on the leftmost edge of the index, where we find a key value k_1. We assign this scan a search key of the form $[= k_1]$. After fetching the first tuple with key value k_1, on a subsequent request for a tuple we open a second index scan with search key $[> k_1]$, in order to quickly find the next group in the table. When we find this value, k_2, we change the second scan's search key to be $[= k_2]$, and return the tuple that was found. We repeat this procedure for subsequent requests until we have a value k_n such that a search key $[> k_n]$ returns no tuples. At this point, we satisfy requests for tuples by fetching from the scans $[= k_1]$, ..., $[= k_n]$ in a (possibly weighted) round-robin fashion.

Striding can be done without an index as well. We are currently developing a Heap Stride access method, for which we briefly give intuitions here. Although heap striding scans a relation sequentially, it does not necessarily output tuples as soon as they are scanned. This is done because of upstream processing (joins, computation, etc.) that may need to occur before the tuple is delivered to the user — fairness dictates that time for such processing not be spent on a group until it is that group's turn.

Like index striding, heap striding opens a cursor per group as new groups are encountered. The distinction is that this is done while sequentially scanning a file (or "heap", in database terminology.) To begin, a cursor is opened at the start of the heap, and the first tuple is returned, say with group 'A'. The cursor is moved forward in the file *without returning any tuples* until a tuple from a different group is found, say group 'B'. The location of each additional 'A' tuple encountered while looking for a 'B' is enqueued on a "to-do" list associated with the 'A' cursor. This process continues until either (1) all groups have been encountered, or (2) it is considered beneficial to perform a round-robin tour of the existing groups before searching further for new groups. Decision (2) is made based on stored statistics, number of available buffers, upstream processing overheads, and other factors affecting the costs and benefits of returning tuples vs. further exploration of the heap. Round-robin processing is done by visiting each cursor and either fetching the next tuple in its to-do list, or (if the list is empty) exploring the heap file further as described above. A picture of fair round-robin processing is shown in Figure 2; the resulting output pattern would be "`ABCABCABCABCABC...`".

We believe we can realize a heap striding access method to provide efficient, index-free strided access. With intelligent buffering it can still be possible to perform a single I/O per block of a file during heap striding, so the postponement of tuple delivery may not produce associated I/O costs. Many design issues remain in determining policies for managing buffers and to-do lists, and in making tradeoffs between overall efficiency and meeting user goals on relative rates for different groups.

[1] Index striding is naturally applicable to other types of indices as well, but we omit discussion here due to space constraints.



Nested Loops Join (a) | Diagonal Traversal (b) | Ever Larger Squares 1 (c) | Ever Larger Squares 2 (d)

Figure 3: Join Matrix Traversals

3.3 Indexes and Compression

Indexes can be exploited in a variety of ways for online processing. We have seen how they can be used for striding access. They can also be used more directly to provide refining estimations of a dataset.

Consider the root of a B+-tree index. It contains a set of keys $k_1, \ldots, k_n$ which partition the dataset into $n+1$ buckets of roughly equal size. This can be thought of as a representation of the distribution of values in the indexed column. In fact, any level of depth d in the tree is a rough equi-depth histogram of $(n+1)^d$ buckets. Thus a standard B+-tree can be traversed in *breadth-first* order to produce an iteratively refining estimate of a data set [Ant93].

Two-dimensional indexes like R-trees can be used to iteratively refine graphical or geographical datasets, and are perfectly suited for online data visualization. Compression techniques like wavelets are commonly used in online network delivery of images. It happens that the standard Haar wavelet encoding [Fal96] is extremely close to the traditional quad-tree [FB74], which is in turn isomorphic to the Z-ordering of Orenstein [Ore86]. This analogy motivates and justifies the use of multi-dimensional indexes for online delivery of data, and conversely suggests interesting directions for online indexing and search, using the wavelet estimation described above to model the eventual state of an n-dimensional search tree.

More flexible trees can be used for online processing yet more effectively. Ranked trees [Olk93] and pseudo-ranked trees [Ant92] allow more accurate “bucket depth” information to be included with the keys. Generalized Search Trees (GiSTs) allow essentially arbitrary flexibility in keys [HNP95]; Aoki has proposed extensions to the GiST framework [Aok97] to encompass statistical (non-restrictive) keys and arbitrary traversal patterns.

3.4 Plane-Sweep Joins

Haas has recently noted that nested loops join is not necessarily appropriate for online estimation; this is because one can tighten a confidence interval only once per tuple of the outer relation [Haa97b]. As an alternative, in joint work we are considering joins which draw on ever larger combinations of tuples from the two input relations. If we picture the tuples in the two relations being laid out as axes of a matrix, nested-loops join visits entries in the matrix in row-major fashion, and confidence intervals shrink at the end of each row (Figure 3a). As an alternative, we propose sweeping the matrix diagonally (Figure 3b), or via ever-larger squares (Figure 3c, 3d). For such traversals, confidence intervals can be shrunk every time a square submatrix containing (1,1) is completely traversed. We are currently investigating efficient versions of such joins.

3.5 Competition and Multi-Threading

Sometimes it can be difficult to predict the processing scheme that will produce the best behavior. Antoshenkov has proposed running multiple access methods in competition to resolve this problem in standard relational systems [Ant93, AZ96]. This approach is particularly important in online processing: one processing scheme may produce online results quickly, while another may produce final answers far more quickly than the first. In such a scenario it can make sense to execute both schemes, letting the first one produce estimates until the second is complete.

A similar theme of multi-threading is useful in a non-competitive manner. In a multi-phase algorithm like mining association rules, user modifications at one phase can often be propagated through later phases differentially. To make this work in an online setting, the original algorithm and the user modification should each represent a thread of processing — it is both correct and beneficial to execute many such threads simultaneously, to encourage online feedback to user input as well as to promote progress toward completion [ASY97]. Scheduling of the different threads should be handled carefully to maximize feedback throughput. It may make sense in some scenarios to allow users to adjust the thread scheduling via relative speed controls.

4 Conclusion

It is our belief that online processing is so natural as to require little further motivation. After a brief digression into the economics of computing and the art of benchmarking, we conclude with some side benefits of research into online processing.

4.1 Online Economics and Benchmarking

Typical cost/performance benchmarks (TPC benchmarks, Dollar Sort, etc.) compare the cost and speed of different hardware/software solutions. Analogies are sometimes made between these benchmarks and auto racing (e.g., the "Indy" and "Daytona" versions of sorting benchmarks [NBC+94]). If we apply the logic of such benchmarks to automobiles, we are led to buy *the fastest car we can purchase within our budget.*

Few of us use such logic in choosing a car. A more reasonable way to choose a car is to buy the most reliable car that matches our budget and functionality needs. Bringing the analogy back to computing, we really want a computing system that provides quick, reliable answers at a reasonable price. We should be willing to sacrifice some negligible reliability to save money. By this argument, a single PC running online software may be far more cost-effective than the price/performance winner of a TPC benchmark winner at a particular budget — the benchmark winner consumes enormous resources to compute a 100% accurate answer, whereas a 99% accurate answer is available to a slower system in the same amount of time. Most consumers will choose the Ford over the Mercedes.

4.2 Subsidiary Benefits of Online Processing

We have seen that by stressing interactivity over speed, online processing allows us to provide far cheaper, more usable solutions to computing challenges. This research has some additional benefits as well:

- Online processing is a natural and long-sought [BDD+89, SSU90, SAD+93] meeting point for research in databases and user interfaces.
- The current "database dinosaurs" have no clear advantage in developing online processing systems. This is an inherently lightweight systems domain, and the field is open.

- Online systems provide "crystal ball" rather than "black box" behavior, allowing users to predict and react to the system output rather than play time-consuming guessing games using trial-and-error or "relevance feedback". This is especially important for IR and AI-based systems whose ill-defined semantics have to be guessed at by users.

- Online systems make impossible queries possible. This is no surprise to statisticians, census-takers and pollsters who can predict the outcome of events before they complete (or even before they start!). There is no reason we cannot do the same in software.

A variety of techniques beyond those described here have been emerging over the last few years; some of them are presented in companion articles in this bulletin. We expect online processing to become an important research and development theme over the next few years. We are currently implementing our online processing techniques and interfaces in the context of the Informix Universal Server and Informix MetaCube Explorer.

Acknowledgments

Thanks to Paul Aoki, Ron Avnur, Marti Hearst, and Allison Woodruff for feedback on an early draft of this paper.

References

[ACSW96] Alexander Aiken, Jolly Chen, Michael Stonebraker, and Allison Woodruff. Tioga-2: A direct-manipulation database visualization environment. In *Proc. 12th IEEE International Conference on Data Engineering*, pages 208–217, New Orleans, February 1996.

[Ant92] Gennady Antoshenkov. Random Sampling from Pseudo-Ranked B+ Trees. In *Proc. 18th International Conference on Very Large Data Bases*, pages 375–382, Vancouver, August 1992.

[Ant93] Gennady Antoshenkov. Query Processing in DEC Rdb: Major Issues and Future Challenges. *IEEE Data Engineering Bulletin*, 16(4):42–52, 1993.

[Aok97] Paul M. Aoki. Generalizing "Search" in Generalized Search Trees. Submitted for publication, June 1997.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, September 1994.

[ASY97] Ron Avnur, Jonathan Spier, and Shu-Yuen Didi Yao. GOLD (GUI OnLine Data) Mining for Association Rules. Class project, CS286, U. C. Berkeley, May 1997.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[BDD+89] Philip A. Bernstein, Umeshwar Dayal, David J. DeWitt, Dieter Gawlick, Jim Gray, Matthias Jarke, Bruce G. Lindsay, Peter C. Lockemann, David Maier, Erich J. Neuhold, Andreas Reuter, Lawrence A. Rowe, Hans-Jorg Schek, Joachim W. Schmidt, Michael Schrefl, and Michael Stonebraker. Future Directions in DBMS Research — The Laguna Beach Report. *ACM SIGMOD Record*, 18(1):17–26, 1989.

[Fal96] Christos Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Publishers, Boston, 1996.

[FB74] R. A. Finkel and J. L. Bentley. Quad-Trees: A Data Structure For Retrieval On Composite Keys. *ACTA Informatica*, 4(1):1–9, 1974.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Technical Report MSR-TR-95-22, Microsoft Research, 1996.

[Haa96] P. J. Haas. Hoeffding Inequalities for Join-Selectivity Estimation and Online Aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1996.

[Haa97a] Peter J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Proc. 9th International Conference on Scientific and Statistical Database Management*, Olympia, WA, August 1997.

[Haa97b] Peter J. Haas. Personal communication. May 1997.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[LRB+97] Miron Livny, Raghu Ramakrishnan, Kevin S. Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, and Jussi Myllymaki. DEVise: Integrated Querying and Visualization of Large Datasets. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 233–242, Minneapolis, May 1994.

[Olk93] Frank Olken. *Random Sampling from Databases.* PhD thesis, University of California, Berkeley, 1993.

[Ore86] J. A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 326–336, Washington, D.C., May 1986.

[PF93] K. Perlin and D. Fox. Pad: An Alternative Approach to the Computer Interface. In *Proc. ACM SIGGRAPH*, pages 57–64, Anaheim, 1993.

[SAD+93] Michael Stonebraker, Rakesh Agrawal, Umeshwar Dayal, Erich J. Neuhold, and Andreas Reuter. Database Research at the Crossroads: The Vienna Update. In *Proc. 19th International Conference on Very Large Data Bases*, pages 688–692, Dublin, August 1993.

[SSU90] Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Systems: Achievements and Opportunities — The "Lagunita" Report. *ACM SIGMOD Record*, 19(4):6–22, 1990.

[ZDN97] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.