Chapter 3: Techniques Everyone Should Know

Introduced by Peter Bailis

Selected Readings:

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. , IBM, September, 1975.

Rakesh Agrawal, Michael J. Carey, Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987, 609-654.

C. Mohan, Bruce G. Lindsay, Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986, 378-396.

In this chapter, we present primary and near-primary sources for several of the most important core concepts in database system design: query planning, concurrency control, database recovery, and distribution. The ideas in this chapter are so fundamental to modern database systems that nearly every mature database system implementation contains them. Three of the papers in this chapter are far and away the canonical references on their respective topics. Moreover, in contrast with the prior chapter, this chapter focuses on broadly applicable techniques and algorithms rather than whole systems.

Query Optimization

Query optimization is important in relational database architecture because it is core to enabling data-independent query processing. Selinger et al.'s foundational paper on System R enables practical query optimization by decomposing the problem into three distinct subproblems: cost estimation, relational equivalences that define a search space, and cost-based search.

The optimizer provides an estimate for the cost of executing each component of the query, measured in terms of I/O and CPU costs. To do so, the optimizer relies on both pre-computed statistics about the contents of each relation (stored in the system catalog) as well as a set of heuristics for determining the cardinality (size) of the query output (e.g., based on estimated predicate selectivity). As an exercise, consider these heuristics in detail: when do they make sense, and on what inputs will they fail? How might they be improved?

Using these cost estimates, the optimizer uses a dynamic programming algorithm to construct a plan for the query. The optimizer defines a set of physical operators that implement a given logical operator (e.g., looking up a tuple using a full 'segment' scan versus an index). Using this set, the optimizer iteratively constructs a "left-deep" tree of operators that in turn uses the cost heuristics to minimize the total amount of estimated work required to run the operators, accounting for "interesting orders" required by upstream consumers. This avoids having to consider all possible orderings of operators but is still exponential in the plan size; as we discuss in Chapter 7, modern query optimizers still struggle with large plans (e.g., many-way joins). Additionally, while the Selinger et al. optimizer performs compilation in advance, other early systems, like Ingres [25] interpreted the query plan – in effect, on a tuple-by-tuple basis.

Like almost all query optimizers, the Selinger et al. optimizer is not actually "optimal" – there is no guarantee that the plan that the optimizer chooses will be the fastest or cheapest. The relational optimizer is closer in spirit to code optimization routines within modern language compilers (i.e., will perform a best-effort search) rather than mathematical optimization routines (i.e., will find the best solution). However, many of today's relational engines adopt the basic methodology from the paper, including the use of binary operators and cost estimation.

Concurrency Control

Our first paper on transactions, from Gray et al., introduces two classic ideas: multi-granularity locking and multiple lock modes. The paper in fact reads as two separate papers.

First, the paper presents the concept of multi-granularity locking. The problem here is simple: given a database with a hierarchical structure, how should we perform mutual exclusion? When should we lock at a coarse granularity (e.g., the whole database) versus a finer granularity (e.g., a single record), and how can we support concurrent access to different portions of the hierarchy at once? While Gray et al.'s hierarchical layout (consisting of databases, areas, files, indexes, and records) differs slightly from that of a modern database system, all but the most rudimentary database locking systems adapt their proposals today.

Second, the paper develops the concept of multiple degrees of isolation. As Gray et al. remind us, a goal of concurrency control is to maintain data that is "consistent" in that it obeys some logical assertions. Classically, database systems used serializable transactions as a means of enforcing consistency: if individual transactions each leave the database in a "consistent" state, then a serializable execution (equivalent to some serial execution of the transactions) will guarantee that all transactions observe a "consistent" state of the database [5]. Gray et al.'s "Degree 3" protocol describes the classic (strict) "two-phase locking" (2PL), which guarantees serializable execution and is a major concept in transaction processing.

However, serializability is often considered too expensive to enforce. To improve performance, database systems often instead execute transactions using non-serializable isolation. In the paper here, holding locks is expensive: waiting for a lock in the case of a conflict takes time, and, in the event of a deadlock, might take forever (or cause aborts). Therefore, as early as 1973, database systems such as IMS and System R began to experiment with non-serializable policies. In a lock-based concurrency control system, these policies are implemented by holding locks for shorter durations. This allows greater concurrency, may lead to fewer deadlocks and system-induced aborts, and, in a distributed setting, may permit greater availability of operation.

In the second half of this paper, Gray et al. provide a rudimentary formalization of the behavior of these lock-based policies. Today, they are prevalent; as we discuss in Chapter 6, non-serializable isolation is the default in a majority of commercial and open source RDBMSs, and some RDBMSs do not offer serializability at all. Degree 2 is now typically called Repeatable Read isolation and Degree 1 is now called Read Committed isolation, while Degree 0 is infrequently used [1]. The paper also discusses the important notion of recoverability: policies under which a transaction can be aborted (or "undone") without affecting other transactions. All but Degree 0 transactions satisfy this property.

A wide range of alternative concurrency control mechanisms followed Gray et al.'s pioneering work on lock-based serializability. As hardware, application demands, and access patterns have changed, so have concurrency control subsystems. However, one property of concurrency control remains a near certainty: there is no unilateral "best" mechanism in concurrency control. The optimal strategy is workload-dependent. To illustrate this point, we've included a study from Agrawal, Carey, and Livny. Although dated, this paper's methodology and broad conclusions remain on target. It's a great example of thoughtful, implementation-agnostic performance analysis work that can provide valuable lessons over time.

Methodologically, the ability to perform so-called "back of the envelope" calculations is a valuable skill: quickly estimating a metric of interest using crude arithmetic to arrive at an answer within an order of magnitude of the correct value can save hours or even years of systems implementation and performance analysis. This is a long and useful tradition in database systems, from the "Five Minute Rule" [12] to Google's "Numbers Everyone Should Know" [4]. While some of the lessons drawn from these estimates are transient [10, 8], often the conclusions provide long-term lessons.

However, for analysis of complex systems such as concurrency control, simulation can be a valuable intermediate step between back of the envelope and full-blown systems benchmarking. The Agrawal study is an example of this approach: the authors use a carefully designed system and user model to simulate locking, restart-based, and optimistic concurrency control.

Several aspects of the evaluation are particularly valuable. First, there is a "crossover" point in almost every graph: there aren't clear winners, as the best-performing mechanism depends on the workload and system configuration. In contrast, virtually every performance study without a crossover point is likely to be uninteresting. If a scheme "always wins," the study should contain an analytical analysis, or, ideally, a proof of why this is the case. Second, the authors consider a wide range of system configurations; they investigate and discuss almost all parameters of their model. Third, many of the graphs exhibit non-monotonicity (i.e., don't always go up and to the right); this a product of thrashing and resource limitations. As the authors illustrate, an assumption of infinite resources leads to dramatically different conclusions. A less careful model that made this assumption implicit would be much less useful.

Finally, the study's conclusions are sensible. The primary cost of restart-based methods is "wasted" work in the event of conflicts. When resources are plentiful, speculation makes sense: wasted work is less expensive, and, in the event of infinite resources, it is free. However, in the event of more limited resources, blocking strategies will consume

fewer resources and offer better overall performance. Again, there is no unilaterally optimal choice. However, the paper's concluding remarks have proven prescient: computing resources are still scarce, and, in fact, few commodity systems today employ entirely restart-based methods. However, as technology ratios – disk, network, CPU speeds – continue to change, re-visiting this trade-off is valuable.

Database Recovery

Another major problem in transaction processing is maintaining durability: the effects of transaction processing should survive system failures. A near-ubiquitous technique for maintaining durability is to perform logging: during transaction execution, transaction operations are stored on fault-tolerant media (e.g., hard drives or SSDs) in a log. Everyone working in data systems should understand how write-ahead logging works, preferably in some detail.

The canonical algorithm for implementing a "No Force, Steal" WAL-based recovery manager is IBM's ARIES algorithm, the subject of our next paper. (Senior database researchers may tell you that very similar ideas were invented at the same time at places like Tandem and Oracle.) In ARIES, the database need not write dirty pages to disk at commit time ("No Force"), and the database can flush dirty pages to disk at any time ("Steal") [15]; these policies allow high performance and are present in almost every commercial RDBMS offering but in turn add complexity to the database. The basic idea in ARIES is to perform crash recovery in three stages. First, ARIES performs an analysis phase by replaying the log forwards in order to determine which transactions were in progress at the time of the crash. Second, ARIES performs a redo stage by (again) replaying the log and (this time) performing the effects of any transactions that were in progress at the time of the crash. Third, ARIES performs an undo stage by playing the log backwards and undoing the effect of uncommitted transactions. Thus, the key idea in ARIES is to "repeat history" to perform recovery; in fact, the undo phase can execute the same logic that is used to abort a transaction during normal operation.

ARIES should be a fairly simple paper but it is perhaps the most complicated paper in this collection. In graduate database courses, this paper is a rite of passage. However, this material is fundamental, so it is important to understand. Fortunately, Ramakrishnan and Gehrke's undergraduate textbook [22] and a survey paper by Michael Franklin [7] each provide a milder treatment. The full ARIES paper we have included here is complicated significantly by its diversionary discussions of the drawbacks of alternative design decisions along the way. On the first pass, we encourage readers to ignore this material and focus solely on the ARIES approach. The drawbacks of alternatives are important but should be saved for a more careful second or third read. Aside from its organization, the discussion of ARIES protocols is further complicated by discussions of managing internal state like indexes (i.e., nested top actions and logical undo logging — the latter of which is also used in exotic schemes like Escrow transactions [20]) and techniques to minimize downtime during recovery. In practice, it is important for recovery time to appear as short as possible; this is tricky to achieve.

Distribution

Our final paper in this chapter concerns transaction execution in a distributed environment. This topic is especially important today, as an increasing number of databases are distributed – either replicated, with multiple copies of data on different servers, or partitioned, with data items stored on disjoint servers (or both). Despite offering benefits to capacity, durability, and availability, distribution introduces a new set of concerns. Servers may fail and network links may be unreliable. In the absence of failures, network communication may be costly.

We concentrate on one of the core techniques in distributed transaction processing: atomic commitment (AC). Very informally, given a transaction that executes on multiple servers (whether multiple replicas, multiple partitions, or both), AC ensures that the transaction either commits or aborts on all of them. The classic algorithm for achieving AC dates to the mid-1970s and is called Two-Phase Commit (2PC; not to be confused with 2PL above!) [9, 18]. In addition to providing a good overview of 2PC and interactions between the commit protocol and the WAL, the paper here contains two variants of AC that improve its performance. The Presumed Abort variant allows processes to avoid forcing an abort decision to disk or acknowledge aborts, reducing disk utilization and network traffic. The Presumed Commit optimization is similar, optimizing space and network traffic when more transactions commit. Note the complexity of the interactions between the 2PC protocol, local storage, and the local transaction manager; the details are important, and correct implementation of these protocols can be challenging.

The possibility of failures substantially complicates AC (and most problems in distributed computing). For example, in 2PC, what happens if a coordinator and participant both fail after all participants have sent their votes but before the coordinator has heard from the failed participant? The remaining participants will not know whether or to commit or abort the transaction: did the failed participant vote YES or vote NO? The participants cannot safely continue. In fact, *any* implementation of AC may block, or fail to make progress, when operating over an unreliable network [2]. Coupled with a serializable concurrency control mechanism, blocking AC means throughput may stall. As a result, a re-

lated set of AC algorithms examined AC under relaxed assumptions regarding both the network (e.g., by assuming a synchronous network) [24] and the information available to servers (e.g., by making use of a "failure detector" that determines when nodes fail) [14].

Finally, many readers may be familiar with the closely related problem of consensus or may have heard of consensus implementations such as the Paxos algorithm. In consensus, any proposal can be chosen, as long as all processes eventually will agree on it. (In contrast, in AC, any individual participant can vote NO, after which all participants must abort.) This makes consensus an "easier" problem than AC [13], but, like AC, any implementation of consensus can also block in certain scenarios [6]. In modern distributed databases, consensus is often used as the basis for replication, to ensure replicas apply updates in the same order, an instance of state-machine replication (see Schneider's tutorial [23]). AC is often used to execute transactions that span multiple partitions. Paxos by Lamport [17] is one of the earliest (and most famous, due in part to a presentation that rivals ARIES in complexity) implementations of consensus. However, the Viewstamped Replication [19] and Raft [21], ZAB [16], and Multi-Paxos [3] algorithms may be more helpful in practice. This is because these algorithms implement a distributed log abstraction (rather than a 'consensus object' as in the original Paxos paper).

Unfortunately, the database and distributed computing communities are somewhat separate. Despite shared interests in replicated data, transfer of ideas between the two were limited for many years. In the era of cloud and Internet-scale data management, this gap has shrunk. For example, Gray and Lamport collaborated in 2006 on Paxos Commit [11], an interesting algorithm combining AC and Lamport's Paxos. There is still much to do in this intersection, and the number of "techniques everyone should know" in this space has grown.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley New York, 1987.

[3] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[4] J. Dean. Designs, lessons and advice from building large distributed systems (keynote). In *LADIS*, 2009.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[7] M. J. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*, pages 1–58–1077, 1997.

[8] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*, 2007.

[9] J. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978.

[10] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[11] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, Mar. 2006.

[12] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *SIGMOD*, 1987.

[13] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *WDAG*, 1995.

[14] R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *SRDS*, 1995.

[15] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[16] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[17] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[18] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, 1979.

[19] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT, 2012.

[20] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.

[21] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[22] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[24] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, 1981.

[25] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.