

CAP Twelve Years Later: How the “Rules” Have Changed

Eric Brewer, University of California, Berkeley

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off of all three.

In the decade since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. The NoSQL movement also has applied it as an argument against traditional databases.

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

- **consistency (C)** equivalent to having a single up-to-date copy of the data;
- **high availability (A)** of that data (for updates); and
- **tolerance to network partitions (P).**

This expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The “2 of 3” formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nu-

ances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

WHY “2 OF 3” IS MISLEADING

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P. The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A. In some sense, the NoSQL movement is about creating choices that focus on availability first and consistency second; databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite. The “ACID, BASE, and CAP” sidebar explains this difference in more detail.

ACID, BASE, AND CAP

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum. The ACID properties focus on consistency and are the traditional approach of databases. My colleagues and I created BASE in the late 1990s to capture the emerging design approaches for high availability and to make explicit both the choice and the spectrum. Modern large-scale wide-area systems, including the cloud, use a mix of both approaches.

Although both terms are more mnemonic than precise, the BASE acronym (being second) is a bit more awkward: Basically Available, Soft state, Eventually consistent. Soft state and eventual consistency are techniques that work well in the presence of partitions and thus promote availability.

The relationship between CAP and ACID is more complex and often misunderstood, in part because the C and A in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees. The four ACID properties are:

Atomicity (A). All systems benefit from atomic operations. When the focus is availability, both sides of a partition should still use atomic operations. Moreover, higher-level atomic operations (the kind that ACID implies) actually simplify recovery.

Consistency (C). In ACID, the C means that a transaction preserves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single-copy consistency, a strict subset of ACID consistency. ACID consistency also cannot be maintained across partitions—partition recovery will need to restore ACID consistency. More generally, maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery.

Isolation (I). Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition. Serializability requires communication in general and thus fails across partitions. Weaker definitions of correctness are viable across partitions via compensation during partition recovery.

Durability (D). As with atomicity, there is no reason to forfeit durability, although the developer might choose to avoid needing it via soft state (in the style of BASE) due to its expense. A subtle point is that, during partition recovery, it is possible to reverse durable operations that unknowingly violated an invariant during the operation. However, at the time of recovery, given a durable history from both sides, such operations can be detected and corrected. In general, running ACID transactions on each side of a partition makes recovery easier and enables a framework for compensating transactions that can be used for recovery from a partition.

In fact, this exact discussion led to the CAP theorem. In the mid-1990s, my colleagues and I were building a variety of cluster-based wide-area systems (essentially early cloud computing), including search engines, proxy caches, and content distribution systems.¹ Because of both revenue goals and contract specifications, system availability was at a premium, so we found ourselves regularly choosing to optimize availability through strategies such as employing

caches or logging updates for later reconciliation. Although these strategies did increase availability, the gain came at the cost of decreased consistency.

The first version of this consistency-versus-availability argument appeared as ACID versus BASE,² which was not well received at the time, primarily because people love the ACID properties and are hesitant to give them up. The CAP theorem's aim was to justify the need to explore a wider design space—hence the “2 of 3” formulation. The theorem first appeared in fall 1998. It was published in 1999³ and in the keynote address at the 2000 Symposium on Principles of Distributed Computing,⁴ which led to its proof.

As the “CAP Confusion” sidebar explains, the “2 of 3” view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Exploring these nuances requires pushing the traditional way of dealing with partitions, which is the fundamental challenge. Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

CAP-LATENCY CONNECTION

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision—the *partition decision*:

- cancel the operation and thus decrease availability, *or*
- proceed with the operation and thus risk inconsistency.

Retrying communication to achieve consistency, for example, via Paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

CAP CONFUSION

Aspects of the CAP theorem are often misunderstood, particularly the scope of availability and consistency, which can lead to undesirable results. If users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as disconnected operation or offline mode,¹ is becoming increasingly important. Some HTML5 features—in particular, on-client persistent storage—make disconnected operation easier going forward. These systems normally choose A over C and thus must recover from long partitions.

Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available. Paxos and atomic multicast systems typically match this scenario.² In Google, the primary partition usually resides within one datacenter; however, Paxos is used on the wide area to ensure global consensus, as in Chubby,³ and highly available durable storage, as in Megastore.⁴

Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible.

Does choosing consistency and availability (CA) as the “2 of 3” make sense? As some researchers correctly point out, exactly what it means to forfeit P is unclear.^{5,6} Can a designer choose not to have partitions? If the choice is CA, and then there is a partition, the choice must revert to C or A. It is best to think about this probabilistically: choosing CA should mean that the probability of a partition is far less than that of other systemic failures, such as disasters or multiple simultaneous faults.

Such a view makes sense because real systems lose both C and A under some sets of faults, so all three properties are a matter of degree.

Thus, pragmatically, a partition is a time bound on communication. Failing to achieve consistency within the time bound implies a partition and thus a choice between C and A for this operation. These concepts capture the core design issue with regard to latency: are two sides moving forward without communication?

This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is that nodes can detect a partition and enter a *partition mode*—a central part of optimizing C and A.

Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

Sometimes it makes sense to forfeit strong C to avoid the high latency of maintaining consistency over a wide area. Yahoo’s PNUTS system incurs inconsistency by maintain-

In practice, most groups assume that a datacenter (single site) has no partitions within, and thus design for CA within a single site; such designs, including traditional databases, are the pre-CAP default. However, although partitions are less likely within a datacenter, they are indeed possible, which makes a CA goal problematic. Finally, given the high latency across the wide area, it is relatively common to forfeit perfect consistency across the wide area for better performance.

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system’s invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

References

1. J. Kistler and M. Satyanarayanan, “Disconnected Operation in the Coda File System,” *ACM Trans. Computer Systems*, Feb. 1992, pp. 3-25.
2. K. Birman, Q. Huang, and D. Freedman, “Overcoming the ‘D’ in CAP: Using Isis2 to Build Locally Responsive Cloud Services,” *Computer*, Feb. 2011, pp. 50-58.
3. M. Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems,” *Proc. Symp. Operating Systems Design and Implementation (OSDI 06)*, Usenix, 2006, pp. 335-350.
4. J. Baker et al., “Megastore: Providing Scalable, Highly Available Storage for Interactive Services,” *Proc. 5th Biennial Conf. Innovative Data Systems Research (CIDR 11)*, ACM, 2011, pp. 223-234.
5. D. Abadi, “Problems with CAP, and Yahoo’s Little Known NoSQL System,” *DBMS Musings*, blog, 23 Apr. 2010; <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>.
6. C. Hale, “You Can’t Sacrifice Partition Tolerance,” 7 Oct. 2010; <http://codahale.com/you-can-t-sacrifice-partition-tolerance>.

ing remote copies asynchronously.⁵ However, it makes the master copy local, which decreases latency. This strategy works well in practice because single user data is naturally partitioned according to the user’s (normal) location. Ideally, each user’s data master is nearby.

Facebook uses the opposite strategy:⁶ the master copy is always in one location, so a remote user typically has a closer but potentially stale copy. However, when users update their pages, the update goes to the master copy directly as do all the user’s reads for a short time, despite higher latency. After 20 seconds, the user’s traffic reverts to the closer copy, which by that time should reflect the update.

MANAGING PARTITIONS

The challenging case for designers is to mitigate a partition’s effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

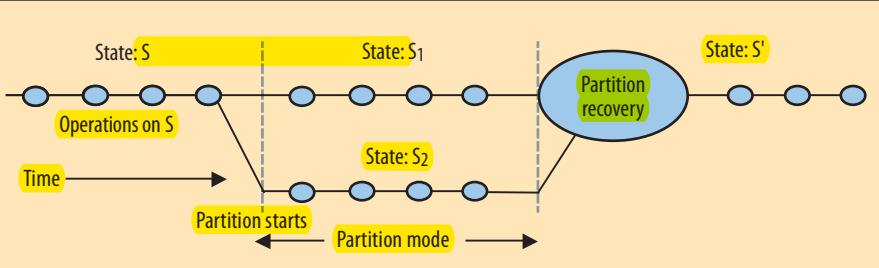


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2 , which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.

- detect the start of a partition,
- enter an explicit partition mode that may limit some operations, and
- initiate partition recovery when communication is restored.

The last step aims to restore consistency and compensate for mistakes the program made while the system was partitioned.

Figure 1 shows a partition's evolution. Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. In such cases, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could have inconsistent operations, it must enter partition mode. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Systems that support disconnected operation clearly have a notion of partition mode, as do some atomic multicast systems, such as Java's JGroups.

Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

Which operations should proceed?

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recov-

ery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant.

For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. (In general, there is no way to tell if the operation will actually violate the invariant, since

the state of the other side is not knowable.) Externalized events, such as charging a credit card, often work this way. In this case, the strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing state, and there is little downside to delaying the operation until the partition ends. The designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition. Bayou's calendar application, for example, shows potentially inconsistent (tentative) entries in a different color.⁷ Such notifications are regularly visible both in workflow applications, such as commerce with e-mail notifications, and in cloud services with an offline mode, such as Google Docs.

One reason to focus on explicit atomic operations, rather than just reads and writes, is that it is vastly easier to analyze the impact of higher-level operations on invariants. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation. In practice, these decisions can also depend on the known state, on the arguments, or on both. For example, in systems with a home node for certain data,⁵ operations can typically proceed on the home node but not on other nodes.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given two versions of an object, A and B, A is newer than B if, for every node in common in their vectors, A's times

are greater than or equal to B's and at least one of A's times is greater.

If it is impossible to order the vectors, then the updates were concurrent and possibly inconsistent. Thus, given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. Recent work⁸ proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

Partition recovery

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually violated invariants and what results were externalized, including the responses sent to the user. The designer must solve two hard problems during recovery:

- the state on both sides must become consistent, and
- there must be compensation for the mistakes made during partition mode.

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. Bayou did this explicitly by rolling back the database to a correct time and replaying the full set of operations in a well-defined, deterministic order so that all nodes reached the same state.⁹ Similarly, source-code control systems such as the Concurrent Versioning System (CVS) start from a shared consistent point and roll forward updates to merge branches.

Most systems cannot always merge conflicts. For example, CVS occasionally has conflicts that the user must resolve manually, and wiki systems with offline mode typically leave conflicts in the resulting document that require manual editing.¹⁰

Conversely, some systems can *always* merge conflicts by choosing certain operations. A case in point is text editing in Google Docs,¹¹ which limits operations to applying a style and adding or deleting text. Thus, although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy.

Using commutative operations is the closest approach to a general framework for automatic state convergence.

The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is commutative, but addition with a bounds check is not (a zero balance, for example).

Designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery.

Recent work by Marc Shapiro and colleagues at INRIA^{12,13} has greatly improved the use of commutative operations for state convergence. The team has developed *commutative replicated data types* (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to

- ensure that all operations during a partition are commutative, or
- represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice.

The latter approach converges state by moving to the maximum of each side's values. It is a formalization and improvement of what Amazon does with its shopping cart:¹⁴ after a partition, the converged value is the union of the two carts, with union being a monotonic set operation. The consequence of this choice is that deleted items may reappear.

However, CRDTs can also implement partition-tolerant sets that both add and delete items. The essence of this approach is to maintain two sets: one each for the added and deleted items, with the difference being the set's membership. Each simplified set converges, and thus so does the difference. At some point, the system can clean things up simply by removing the deleted items from both sets. However, such cleanup generally is possible only while the system is not partitioned. In other words, the designer must prohibit or postpone some operations during a partition, but these are cleanup operations that do not limit perceived availability. Thus, by implementing state through CRDTs, a designer can choose A and still ensure that state converges automatically after a partition.

Compensating for mistakes

In addition to computing the postpartition state, there is the somewhat harder problem of fixing mistakes made

COMPENSATION ISSUES IN AN AUTOMATED TELLER MACHINE

In the design of an automated teller machine (ATM), strong consistency would appear to be the logical choice, but in practice, A trumps C. The reason is straightforward enough: higher availability means higher revenue. Regardless, ATM design serves as a good context for reviewing some of the challenges involved in compensating for invariant violations during a partition.

The essential ATM operations are deposit, withdraw, and check balance. The key invariant is that the balance should be zero or higher. Because only withdraw can violate the invariant, it will need special treatment, but the other two operations can always execute.

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using stand-in mode (partition mode), modern ATMs limit the net withdrawal to at most k , where k might be \$200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but compensation can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment. Given that the risk is bounded, the problem is not severe. However, suppose that the balance was below zero at some point during the partition (unknown to the ATM), but that a later deposit brought it back up. In this case, the bank might still charge an overdraft fee retroactively, or it might ignore the violation, since the customer has already made the necessary payment.

In general, because of communication delays, the banking system depends not on consistency for correctness, but rather on auditing and compensation. Another example of this is “check kiting,” in which a customer withdraws money from multiple branches before they can communicate and then flees. The overdraft will be caught later, perhaps leading to compensation in the form of legal action.

during partitioning. The tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated, which in turn enables the designer to create a restoration strategy for each such invariant. Typically, the system discovers the violation during recovery and must implement any fix at that time.

There are various ways to fix the invariants, including trivial ways such as “last writer wins” (which ignores some updates), smarter approaches that merge operations, and human escalation. An example of the latter is airplane overbooking: boarding the plane is in some sense partition recovery with the invariant that there must be at least as many seats as passengers. If there are too many passengers, some will lose their seats, and ide-

ally customer service will compensate those passengers in some way.

The airplane example also exhibits an externalized mistake: if the airline had not said that the passenger had a seat, fixing the problem would be much easier. This is another reason to delay risky operations: at the time of recovery, the truth is known. The idea of compensation is really at the core of fixing such mistakes; designers must create compensating operations that both restore an invariant and more broadly correct an externalized mistake.

Technically, CRDTs allow only *locally* verifiable invariants—a limitation that makes compensation unnecessary but that somewhat decreases the approach’s power. However, a solution that uses CRDTs for state convergence could allow the temporary violation of a global invariant, converge the state after the partition, and then execute any needed compensations.

Recovering from externalized mistakes typically requires some history about externalized outputs. Consider the drunk “dialing” scenario, in which a person does not remember making various telephone calls while intoxicated the previous night. That person’s state in the light of day might be sound, but the log still shows a list of calls, some of which might have been mistakes. The calls are the external effects of the person’s state (intoxication). Because the person failed to remember the calls, it could be hard to compensate for any trouble they have caused.

In a machine context, a computer could execute orders twice during a partition. If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates. If externalized, one compensation strategy would be to autogenerate an e-mail to the customer explaining that the system accidentally executed the order twice but that the mistake has been fixed and to attach a coupon for a discount on the next order. Without the proper history, however, the burden of catching the mistake is on the customer.

Some researchers have formally explored compensating transactions as a way to deal with long-lived transactions.^{15,16} Long-running transactions face a variation of the partition decision: is it better to hold locks for a long time to ensure consistency, or release them early and expose uncommitted data to other transactions but allow higher concurrency? A typical example is trying to update all employee records as a single transaction. Serializing this transaction in the normal way locks all records and prevents concurrency. Compensating transactions take a different approach by breaking the large transaction into a saga, which consists of multiple subtransactions, each of which commits along the way. Thus, to abort the larger transaction, the system must undo each already committed subtransaction by issuing a new transaction that corrects for its effects—the compensating transaction.

In general, the goal is to avoid aborting other trans-

actions that used the incorrectly committed data (no cascading aborts). The correctness of this approach depends not on serializability or isolation, but rather on the net effect of the transaction sequence on state and outputs. That is, after compensations, does the database essentially end up in a place equivalent to where it would have been had the subtransactions never executed? The equivalence must include externalized actions; for example, refunding a duplicate purchase is hardly the same as not charging that customer in the first place, but it is arguably equivalent. The same idea holds in partition recovery. A service or product provider cannot always undo mistakes directly, but it aims to admit them and take new, compensating actions. How best to apply these ideas to partition recovery is an open problem. The “Compensation Issues in an Automated Teller Machine” sidebar describes some of the concerns in just one application area.

System designers should not blindly sacrifice consistency or availability when partitions exist. Using the proposed approach, they can optimize both properties through careful management of invariants during partitions. As newer techniques, such as version vectors and CRDTs, move into frameworks that simplify their use, this kind of optimization should become more widespread. However, unlike ACID transactions, this approach requires more thoughtful deployment relative to past strategies, and the best solutions will depend heavily on details about the service’s invariants and operations. □

Acknowledgments

I thank Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao, and the IEEE Computer Society volunteers for their helpful feedback on this work.

References

1. E. Brewer, “Lessons from Giant-Scale Services,” *IEEE Internet Computing*, July/Aug. 2001, pp. 46-55.
2. A. Fox et al., “Cluster-Based Scalable Network Services,” *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox and E.A. Brewer, “Harvest, Yield and Scalable Tolerant Systems,” *Proc. 7th Workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, “Towards Robust Distributed Systems,” *Proc. 19th Ann. ACM Symp. Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10; www.cs.berkeley.edu/~brewer/PODC2000.pdf.
5. B. Cooper et al., “PNUTS: Yahoo!’s Hosted Data Serving Platform,” *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
6. J. Sobel, “Scaling Out,” *Facebook Engineering Notes*, 20 Aug. 2008; www.facebook.com/note.php?note_id=23844338919&id=9445547199.
7. W. K. Edwards et al., “Designing and Implementing Asynchronous Collaborative Applications with Bayou,” *Proc. 10th Ann. ACM Symp. User Interface Software and Technology* (UIST 97), ACM, 1999, pp. 119-128.
8. P. Mahajan, L. Alvisi, and M. Dahlin, *Consistency, Availability, and Convergence*, tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011.
9. D.B. Terry et al., “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System,” *Proc. 15th ACM Symp. Operating Systems Principles* (SOSP 95), ACM, 1995, pp. 172-182.
10. B. Du and E.A. Brewer, “DTWiki: A Disconnection and Intermittency Tolerant Wiki,” *Proc. 17th Int’l Conf. World Wide Web* (WWW 08), ACM, 2008, pp. 945-952.
11. “What’s Different about the New Google Docs: Conflict Resolution,” blog; http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_22.html.
12. M. Shapiro et al., “Conflict-Free Replicated Data Types,” *Proc. 13th Int’l Conf. Stabilization, Safety, and Security of Distributed Systems* (SSS 11), ACM, 2011, pp. 386-400.
13. M. Shapiro et al., “Convergent and Commutative Replicated Data Types,” *Bulletin of the EATCS*, no. 104, June 2011, pp. 67-88.
14. G. DeCandia et al., “Dynamo: Amazon’s Highly Available Key-Value Store,” *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.
15. H. Garcia-Molina and K. Salem, “SAGAS,” *Proc. ACM SIGMOD Int’l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259.
16. H. Korth, E. Levy, and A. Silberschatz, “A Formal Approach to Recovery by Compensating Transactions,” *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pp. 95-106.

Eric Brewer is a professor of computer science at the University of California, Berkeley, and vice president of infrastructure at Google. His research interests include cloud computing, scalable servers, sensor networks, and technology for developing regions. He also helped create USA.gov, the official portal of the federal government. Brewer received a PhD in electrical engineering and computer science from MIT. He is a member of the National Academy of Engineering. Contact him at brewer@cs.berkeley.edu.

ICDE 2012

28th IEEE International Conference on Data Engineering

1-5 April 2012

Washington, DC, USA

ICDE addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications.

Learn more at:

<http://www.icde12.org/>





Bases de datos

CAP

ACID and BASE

Introducción a bases de dato relacionales

Normalización

Semántica

Almacenamiento

NULLS

Tuplas espúreas

Dependencias funcionales (DF)

Formas normales 1FN

Formas normales 2FN

Formas normales 3FN

NoSQL

Desnormalizacion

El mundo de los sistemas de bases de datos

Bases de datos relacionales

Sistemas más y más pequeños

Integración de la información

Overview de los sistemas de administración de bases de datos DBMS

Comandos del lenguaje de definición de datos

Descripción general del procesamiento de consultas

Almacenamiento y buffer management

Procesamiento transaccional

El procesador de consultas

Bases de datos de grafos

Optimización

Wide column (Cassandra)

Control de concurrencia

CAP

- El teorema CAP establece que cualquier sistema de datos compartidos en red puede tener como máximo dos de tres propiedades deseables:
 - coherencia/consistencia (C) equivalente a tener una única copia actualizada de los datos.
 - Alta disponibilidad (A) de esos datos para actualizaciones. si se puede comunicar con un nodo en el cluster, entonces se pueden leer y escribir dato.
 - Tolerancia a las particiones de la red (P). El cluster puede sobrevivir a roturas de comunicación que lo dividan en particiones que no pueden comunicarse entre ellas
- Aunque los diseñadores todavía necesitan elegir entre coherencia y disponibilidad cuando hay particiones presentes, existe una increíble variedad de flexibilidad para manejar particiones y recuperarse de ellas. CAP moderno debería ser maximizar las combinaciones de consistencia y disponibilidad que tengan sentido para la aplicación específica. Este enfoque incorpora planes de

operación durante una partición y de recuperación posterior, ayudando así a los diseñadores a pensar en CAP más allá de sus limitaciones históricamente percibidas.

- La forma más sencilla de entender CAP es pensar en dos nodos en lados opuestos de una partición. Permitir que al menos un nodo actualice el estado hará que los nodos se vuelvan inconsistentes, perdiendo así C. Del mismo modo, si la elección es preservar la coherencia, un lado de la partición debe actuar como si no estuviera disponible, perdiendo así A. Sólo cuando los nodos comunicar es posible preservar tanto la coherencia como la disponibilidad, perdiendo así P. La creencia general es que, para los sistemas de área amplia, los diseñadores no pueden renunciar a P y, por lo tanto, tienen una elección difícil entre C y A. En cierto sentido, el movimiento NoSQL se trata de crear opciones que se centren primero en la disponibilidad y en segundo lugar en la coherencia; las bases de datos que se adhieren a las propiedades de ACID (atomicidad, consistencia, aislamiento y durabilidad) hacen lo contrario.
- Debido a que las particiones son raras, CAP debería permitir C y A perfectas la mayor parte del tiempo, pero cuando las particiones están presentes o se perciben, es necesaria una estrategia que las detecte y las tenga en cuenta explícitamente. Esta estrategia debe tener tres pasos: detectar particiones, ingresar a un modo de partición explícito que pueda limitar algunas operaciones e iniciar un proceso de recuperación para restaurar la coherencia y compensar los errores cometidos durante una partición.
- Operacionalmente, la esencia de CAP tiene lugar durante un timeout, un período en el que el programa debe tomar una decisión fundamental: la decisión de partición: cancelar la operación y así disminuir la disponibilidad o continuar con la operación y así correr el riesgo de inconsistencia. En algún momento el programa debe tomar la decisión; Reintentar la comunicación indefinidamente es, en esencia, elegir C en lugar de A.
- Algunos aspectos del teorema CAP a menudo se malinterpretan. En la práctica, la mayoría de los grupos asumen que un centro de datos (sitio único) no tiene el alcance de la disponibilidad y la coherencia, lo que puede conducir a resultados no deseados. Si los usuarios no pueden acceder al servicio en absoluto, no pueden elegir entre C y A excepto cuando parte del servicio se ejecuta en el cliente. Esta excepción, comúnmente conocida como funcionamiento desconectado o modo fuera de línea, está adquiriendo cada vez más importancia. Algunas características de HTML5 (en particular, el almacenamiento persistente en el cliente) facilitan la operación desconectada en el futuro. Estos sistemas normalmente eligen A en lugar de C y, por lo tanto, deben recuperarse de particiones largas.
- Finalmente, esta visión significa que los diseñadores pueden establecer límites de tiempo intencionalmente de acuerdo con los tiempos de respuesta objetivo; Los sistemas con límites más estrictos probablemente entrarán en modo de partición con más frecuencia y en momentos en que la red es simplemente lenta y no está realmente particionada.
- Luego de una partición, el sistema debe restaurar la consistencia y compensar por los errores que el programa haya hecho durante la partición.
- Por ejemplo, para la invariante de que las claves de una tabla son únicas, los diseñadores normalmente deciden arriesgar esa invariante y permitir claves duplicadas durante una partición. Las claves duplicadas son fáciles de detectar durante la recuperación y, suponiendo que se puedan fusionar, el diseñador puede restaurar fácilmente la invariante. Sin embargo, para una invariante que debe mantenerse durante una partición, el diseñador debe prohibir o modificar operaciones que puedan violarla (en general, no hay manera de saber si la operación realmente violará el invariante, ya que el estado del otro lado no se puede conocer). Los eventos externalizados, como cargar una

tarjeta de crédito, a menudo funcionan de esta manera. En este caso, la estrategia es registrar la intención y ejecutarla después de la recuperación. Estas transacciones suelen ser parte de un flujo de trabajo más amplio que tiene un estado explícito de procesamiento de pedidos, y hay pocas desventajas en retrasar la operación hasta que finalice la partición. El diseñador pierde A de una manera que los usuarios no ven. Los usuarios sólo saben que realizaron un pedido y que el sistema lo ejecutará más tarde. De manera más general, el modo de partición genera un desafío fundamental en la interfaz de usuario, que es comunicar que las tareas están en progreso pero no completadas.

- En algún momento, la comunicación se reanuda y la partición finaliza. Durante la partición, cada lado estuvo disponible y, por lo tanto, avanzó, pero la partición retrasó algunas operaciones y violó algunas invariantes. En este punto, el sistema conoce el estado y el historial de ambas partes porque mantuvo un registro cuidadoso durante el modo de partición. El estado es menos útil que el historial, a partir del cual el sistema puede deducir qué operaciones realmente violaron las invariantes y qué resultados se externalizaron, incluidas las respuestas enviadas al usuario. El diseñador debe resolver dos problemas difíciles durante la recuperación:
 - el Estado de ambas partes debe volverse coherente, y
 - debe haber compensación por los errores cometidos durante el modo de partición.

Generalmente es más fácil arreglar el estado actual comenzando desde el estado en el momento de la partición y avanzando ambos conjuntos de operaciones de alguna manera, manteniendo un estado consistente a lo largo del camino.

- **Las 8 Falacias de la Computación Distribuida**

1. La red es confiable
 2. La latencia es cero; la latencia no es problema
 3. El ancho de banda es infinito
 4. La red es segura
 5. La topología no cambia
 6. Hay uno y solo un administrador
 7. El coste de transporte es cero
 8. La red es homogénea
- Las transacciones bancarias son inconsistentes, particularmente para ATMs, que están diseñados para tener un comportamiento en modo normal y otro en modo partición. En modo partición la Availability es elegida por sobre la Consistencia. Históricamente la industria financiera ha sido inconsistente a causa de la imperfección de las comunicaciones. ATMs realizan operaciones conmutativas de manera que el orden en el cuales se aplican no importa. Si hay una partición el cajero puede seguir funcionando; luego al volver a recuperar la partición se envían las operaciones y el saldo final sigue siendo correcto. Lo que se hace es delimitar y administrar el riesgo. Los ATMs son rentables aún a costa de alguna pérdida.

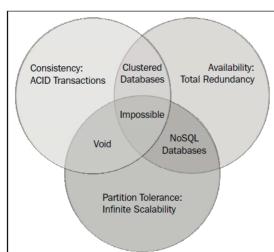
ACID and BASE

- ACID y BASE representan dos filosofías de diseño en extremos opuestos del espectro de consistencia-disponibilidad. Las propiedades ACID se centran en la coherencia y son el enfoque tradicional de las bases de datos. BASE se creó a finales de los años 1990 para captar los enfoques de diseño emergentes para la alta disponibilidad y hacer explícitos tanto la elección como el

espectro. Los sistemas modernos de gran área y gran escala, incluida la nube, utilizan una combinación de ambos enfoques.

- Aunque ambos términos son más mnemotécnicos que precisos, el acrónimo BASE (que ocupa el segundo lugar) es un poco más incómodo: Basically Available Soft State. Soft state y consistencia eventual, son técnicas que funcionan bien en presencia de particiones y, por lo tanto, promueven la disponibilidad.
- En BASE:
 - Disponibilidad básica: cada solicitud tiene garantizada una respuesta de ejecución exitosa/fallida.
 - Estado suave: el estado del sistema puede cambiar con el tiempo, a veces sin ninguna intervención (para una eventual consistencia).
 - Consistencia eventual: la base de datos puede ser momentáneamente inconsistente, pero eventualmente lo será.
- Propiedades ACID:
 - Atomicidad: se ejecuta toda la transacción o nada (si hubo algún error). Todos los sistemas se benefician de las operaciones atómicas. Cuando el foco es la disponibilidad, ambos lados de una partición aún deben usar operaciones atómicas. Además, las operaciones atómicas de alto nivel (del tipo que implica ACID) en realidad simplifican la recuperación.
 - Consistencia: Se espera que las transacciones preserven la coherencia de la base de datos. Es básicamente un contrato entre los procesos y el almacén de datos. Este contrato dice que, si los procesos aceptan obedecer ciertas reglas, el almacén promete funcionar correctamente. Todos ven los mismos datos al mismo tiempo. Significa que una transacción conserva todas las reglas de la base de datos, como las claves únicas. Por el contrario, la C en CAP se refiere sólo a la coherencia de una sola copia, un subconjunto estricto de la coherencia ACID. La coherencia ACID tampoco se puede mantener entre particiones; la recuperación de la partición deberá restaurar la coherencia ACID. En términos más generales, mantener invariantes durante las particiones podría ser imposible, de ahí la necesidad de pensar cuidadosamente qué operaciones no permitir y cómo restaurar invariantes durante la recuperación.
 - consistencia fuerte: todas las operaciones de lectura deben devolver datos de la última operación de escritura completada.
 - Consistencia eventual: las operaciones de lectura verán, conforme pasa el tiempo, las escrituras. En un estado de equilibrio, el estado devolvería el último valor escrito.
 - Read Your Own Writes (RYOW): El cliente lee sus actualizaciones inmediatamente después de que hayan sido completadas, independientemente si escribe en un server y lee de otro. Las actualizaciones de otros clientes no tienen por qué ser visibles instantáneamente.
 - Consistencia de sesión: Un poco más débil que RYOW. Provee ésta consistencia solo si el cliente está dentro de la misma sesión. Usualmente sobre el mismo server.
 - aislamiento: el hecho de que cada transacción debe parecer ejecutada como si no se estuviera ejecutando ninguna otra transacción al mismo tiempo. El aislamiento es el núcleo del teorema CAP. Si el sistema requiere aislamiento ACID, puede operar como máximo en un lado durante una partición. La serialización requiere comunicación en general y, por lo tanto, falla entre particiones. Las definiciones más débiles de corrección son viables entre particiones mediante la compensación durante la recuperación de la partición.

- Durabilidad: la condición de que el efecto en la base de datos de una transacción nunca debe perderse, una vez que la transacción se haya completado. Al igual que con la atomicidad, no hay razón para perder la durabilidad, aunque el desarrollador podría optar por evitar su necesidad a través del estado blando (al estilo de BASE) debido a su costo. Un punto sutil es que, durante la recuperación de la partición, es posible revertir operaciones duraderas que, sin saberlo, violaron una invariante durante la operación. Sin embargo, en el momento de la recuperación, dada una historia duradera de ambas partes, dichas operaciones pueden detectarse y corregirse. En general, ejecutar transacciones ACID en cada lado de una partición facilita la recuperación y habilita un marco para compensar transacciones que se pueden utilizar para la recuperación de una partición.



Feature	NoSQL Databases	Relational Databases
Performance	High	Low
Reliability	Poor	Good
Availability	Good	Good
Consistency	Poor	Good
Data Storage	Optimized for huge data	Medium sized to large
Scalability	High	High (but more expensive)

The ACID Properties of Transactions		
Properly implemented transactions are commonly said to meet the "ACID test," where:		
<ul style="list-style-type: none"> • "A" stands for "atomicity," the all-or-nothing execution of transactions. • "I" stands for "isolation," the fact that each transaction must appear to be executed as if no other transaction is executing at the same time. • "D" stands for "durability," the condition that the effect on the database of a transaction must never be lost, once the transaction has completed. 		
The remaining letter, "C," stands for "consistency." That is, all databases have consistency constraints, or expectations about relationships among data elements (e.g., account balances may not be negative after a transaction finishes). Transactions are expected to preserve the consistency of the database.		

Introducción a bases de dato relacionales

- Las cuatro v del bigdata:
 - Volumen: "Research has determined that **a single well can produce about one terabyte of production data – every day**. That must be processed, organized and stored by someone, and can take hours of company time to do so."
 - Variedad: La variedad de información es enorme, video imágenes, audio, texto, música. Se refiere a la gran diferencia que hay en el formato, el origen (las características generales del dato), etc. Hoy en día el dato puede recopilarse en un formato estructurado, fácil de tratar (bases de datos estructuradas, formatos predefinidos, etc.) hasta datos no estructurados extraídos de redes sociales, de videos, etc. La habilidad para saber tratar dicho dato y transformarlo a un formato fácil de analizar definirá la capacidad de dicha empresa de saber gestionar y aprovechar el volumen de datos existente.
 - Velocidad: se refiere a la rapidez a la que se generan los datos en la actualidad. Es, como hemos mencionado, el mayor reto de todos, ya no solo para el departamento de informática, sino para todo resto de departamentos (financiero, marketing, logística, etc.). La clave está en saber recopilar, gestionar, analizar y sobre todo extraer información de negocio de valor en tiempo real. The physical proximity to the exchange server reduces the time from when a firm's buy or sell order is entered and when it's executed. "By co-locating," says Adam Afshar of Hyde Park Global, a high-speed trading firm, "we are able to take 21 milliseconds off our trades. In the past, 21 milliseconds was a trivial matter. Now it's a pivotal matter." Several academic studies have found

that shaving even one millisecond off every trade can be worth \$100 million a year to a large, high-speed trading firm."

- Veracidad: se refiere a la calidad, la predictibilidad y la disponibilidad del dato. Es la variable menos uniforme y menos sencilla de controlar, debido a la dificultad de cerciorarnos de que un dato es 100% fiable. La clave para poder afrontar con garantías esta última faceta del dato es tener un equipo imparcial que ayude a mantener los datos limpios para su posterior evaluación de estrategia de Big Data.
- Qué es una base de datos: Una base de datos es una colección de datos almacenados permanentemente que es:
 - Lógicamente relacionada: los datos se relacionan con otros datos (tablas con tablas).
 - Compartido: muchos usuarios pueden acceder a los datos.
 - Protegido: el acceso a los datos está controlado.
 - Gestionado: los datos tienen integridad y valor.
 - Pueden contener:
 - Tablas: filas y columnas de datos (requieren espacio permanente). Los registros en teoría no deberían estar ni ordenados ni repetidos.
 - Vistas: subconjuntos predefinidos de tablas existentes (no requieren espacio permanente)
 - Triggers (disparadores): declaraciones SQL asociadas a una tabla (no requieren espacio permanente)
 - Stored procedures: programas guardados (requieren espacio permanente)
 - Funciones de usuario definidas: programas que agregan funcionalidad adicional (requieren espacio permanente)
- Los límites de espacio se especifican para cada base de datos y para cada usuario:
 - espacio permanente: cantidad máxima de espacio disponible para tablas e índices subtablas
 - espacio spool: cantidad máxima de espacio de trabajo disponible para solicitudes de procesamiento. Contiene o mantiene resultados intermedios.
- Normalización:
 - Proceso de reducir una estructura de datos compleja a una simple y estable.
 - Implica eliminar atributos, claves y relaciones redundantes del modelo de datos conceptual.
- El procesamiento transaccional es una unidad de trabajo que debe ejecutarse en forma atómica y en aparente aislamiento
 1. Registro para garantizar la durabilidad.
 2. Las transacciones de control de concurrencia deben parecer ejecutarse de forma aislada
 3. Las transacciones de resolución de puntos muertos compiten por los recursos
- Los valores de clave primaria (PK) identifican de forma única cada fila de una tabla. Reglas de clave primaria:
 - Se requiere una clave principal para cada tabla.
 - Sólo se permite una clave principal en una tabla.

- Las claves primarias pueden constar de una o más columnas.
 - Las claves primarias no pueden tener valores duplicados.
 - Las claves primarias no pueden ser nulas.
 - Las claves primarias se consideran valores “no cambiantes”.
- Las claves foráneas (FK) valoran relaciones del modelo.
 - son opcionales.
 - Una tabla puede tener más de una FK
 - puede consistir de más de una columna
 - pueden estar duplicadas
 - pueden ser nulas
 - pueden cambiar
 - Deben existir en otra tabla como PK
- Las ventajas de una base de datos relacional en comparación con otras metodologías de bases de datos incluyen:
 - Más flexible que otros tipos
 - Permite a las empresas responder rápidamente a las condiciones cambiantes.
 - Estar basado en datos vs. impulsado por la aplicación
 - Modelar el negocio, no los procesos
 - Hace que las aplicaciones sean más fáciles de crear porque los datos hacen más trabajo
 - Una sola copia de los datos puede servir para múltiples propósitos
 - Apoyar la tendencia hacia la informática del usuario final
 - Ser fácil de entender
 - No es necesario conocer la ruta de acceso
 - Sólidamente fundamentado en la teoría de conjuntos.
- El motor de parse (parsing engine) es responsable de:
 - realizar el análisis sintáctico desde el punto de vista del sistema.
 - Gestionar sesiones individuales
 - Analizar y optimizar sus solicitudes SQL
 - Creación de planes de consulta con el Optimizador
 - Envío del plan optimizado
 - Enviar la respuesta del conjunto de respuestas al cliente solicitante
- RAID 1 (espejo). Buena performance cuando hay fallas pero alto costo en términos de almacenamiento.
 - cada disco físico tiene exactamente la misma información (copia)
 - reduce el espacio disponible 50%

- RAID 5 (paridad). Reducción de performance cuando hay que reconstruir por fallas en discos, pero más barato en términos de almacenamiento.
 - Datos divididos en 4 discos.
 - Si falla un disco, cualquier bloque faltante se puede reconstruir utilizando los otros tres discos.
 - La paridad reduce el espacio disponible en disco en un 25% en un rango de 4 discos.
 - La reconstrucción de discos fallidos lleva más tiempo que RAID 1.
- El modelo lógico
 - debería ser el mismo sin importar el volumen de datos
 - Los datos se organizan según lo que representan: negocios en el mundo real
 - Es genérico. El modelo lógico es la plantilla para la implementación física en cualquier plataforma RDBMS.

Normalización

- Permite identificar que tablas, columnas y relaciones que se requieren
- Es un enfoque sistemático que permite organizar la información en la base de datos relacional
- Es un proceso de refinado para obtener la mejor estructura posible
- La información se transforma utilizando las formas normales.
- Elimina duplicación de datos innecesaria (no necesariamente toda duplicación)
- Aumenta la consistencia de los datos
- Modela mejor las entidades del mundo real
- Proceso de analizar los esquemas, basándose en dependencias funcionales y claves primarias
- Calidad de Diseño: necesidad de evaluar si una forma de agrupar atributos en un esquema es mejor que otra
- Niveles:
 - Lógico o conceptual: un buen diseño de esquemas a este nivel habilita a los usuarios a entender el significado de los datos de las relaciones
 - Implementación (o de Almacenamiento físico): cómo se almacenan y actualizan las tuplas
- Pautas de diseño. cuatro pautas informales de diseño pueden utilizarse como medida para determinar la calidad de un diseño
 1. Estar seguro que semántica de atributos en esquemas es clara
 2. Reducir la información redundante en tuplas
 3. Reducir la cantidad de valores NULL en tuplas
 4. Desactivar la posibilidad de generar tuplas espúreas.
 5. Minimizar anomalías de inserción, delección y modificación

Semántica

1. Cuanto más fácil es explicar la semántica de los esquemas, mejor es el diseño.
2. No combinar atributos de diversos tipos de entidades y relaciones en una misma relación
3. Diseñar esquemas tal que sea fácil de explicar su significado

Mezcla empleado con proyecto

● Ejemplo.

EMPLEADO_PROJECTO					
E_Nombre	E_DNI	E_Fecha_Nacimiento	Dirección	P_Nombre	P_Número

Ejemplo OK.

EMPLEADO			
E_Nombre	E_DNI	E_Fecha_Nacimiento	Dirección_Laboral
PROYECTO			
P_Nombre	P_Número		
TRABAJA_EN			
E_DNI	P_Número		

Almacenamiento

1. Minimizar espacio de almacenamiento a través del diseño
2. Diseñar esquemas tal que no permitan anomalías de inserción, delección y modificación
3. Si permiten anomalías, señalarlas claramente y asegurar que programas que actualizan BD operarán correctamente
4. Notar que esta pauta puede ser violada en favor de la performance

Diseño A almacena NATURAL JOIN de diseño B

Agrega anomalías de actualización. Almacenar NATURAL JOINS introduce problemas adicionales. anomalías de inserción ,delección y modificación

Si quiero agregar un empleado que no tiene asignado dpto me queda con un null

Si quiero insertar un nuevo dpto que no posee empleados me quedan muchos null (incluido el PK E_DNI)

Si elimino a Marina, perdí al departamento 2 y su info

Modificar un atributo de un departamento, implica modificar todos los registros con ese dpto, caso contrario se generan inconsistencias

Diseño “A”

EMPLEADO..DEPARTAMENTO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
...

Diseño “B”

EMPLEADO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto
Diego	20222333	11/12/1970	5
Laura	33456234	02/04/1985	5
Marina	45432345	23/07/2006	2
Santiago	24345345	18/02/1975	5
...

DEPARTAMENTO

Nro_Depto	D_Nombre
5	Publicidad y Promoción
2	Reclutamiento y Selección

NULLS

1. Atributos no relacionados y agrupados en una misma tabla pueden generar múltiples NULLs en una misma tupla.
2. Evitar asignar atributos a relaciones, cuando estos frecuentemente pueden ser NULLs
3. Si NULLs son inevitables, asegurar que las situaciones son excepcionales y no aplican a la mayoría de las tuplas

Genera problemas:

- Desperdicio espacio almacenamiento
- JOINs (en presencia de NULLs, INNER JOIN produce distinto resultado vs. OUTER JOIN)
- ¿Cómo se interpretan funciones de agregación (COUNT, SUM, etc.)?
- Diversas interpretaciones de NULL

Tuplas espúreas

1. Diseñar esquemas tal que puedan ser relacionados por atributos que se encuentren apropiadamente relacionados por medio de condiciones de igualdad entre ellos (clave primaria, clave foránea), para evitar generación de tuplas espúreas
2. Evitar relaciones que contengan atributos de matching que no sean combinación de claves foránea/primaria porque JOINs sobre ellos pueden producir tuplas espúreas
3. Tengo que partir por la key (si original tiene A,B,C y la key es A), parto en AB y AC.

No deseable. Esta descomposición no es deseable porque cuando se intenta la reconstrucción a través de NATURAL JOIN no se obtiene información correcta

Causa. P_Ubicacion, relaciona ambos esquemas, pero no es ni clave primaria ni clave foránea de ninguno de ellos.

● Descomposición.

E_DNI	E_Fecha_Nacimiento	Nro_PROYECTO	P_Ubicación
20222333	11/12/1970	5	Argentina
33456234	02/04/1985	5	Argentina
45432345	23/07/2006	2	Uruguay
24345345	18/02/1975	5	Argentina

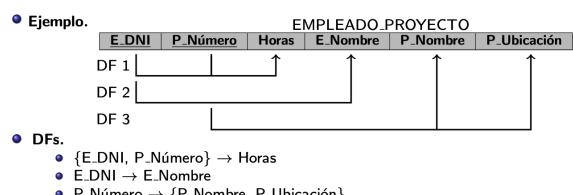
E_Nombre	P_Ubicación
Diego	Argentina
Laura	Argentina
Marina	Uruguay
Santiago	Argentina

- ¿Qué problema genera esta descomposición?
No permite recuperar información original de EMPLEADO.PROYECTO
- ¿Cuál es el resultado de aplicar NATURAL JOIN?
Produce **tuplas espúreas** (información no válida)

E_DNI	E_Fecha_Nacimiento	Nro_PROYECTO	P_Ubicación	E_Nombre
20222333	11/12/1970	5	Argentina	Diego
33456234	02/04/1985	5	Argentina	Diego
24345345	18/02/1975	5	Argentina	Diego

Dependencias funcionales (DF)

- Herramienta formal para el análisis de esquemas. Permite detectar y describir problemas descriptos previamente. Informalmente es la restricción entre dos conjuntos de atributos X e Y de una BD.
- Las dependencias funcionales son propiedad de la semántica (o significado) de los atributos.
- Diseñadores de las BD deben usar su entendimiento de la semántica de atributos de R para especificar las DF y deberá respetar TODOS
- Instancias legales: registro que satisface restricciones de DF se denomina instancia legal, estado legal.
- Inferencia de DF. Dada una relación con sus datos, no es posible determinar sus DF a través de sus valores. Es necesario conocer el significado y relación que existe entre los atributos que la componen
- Existencia. Una DF puede existir si la cumple una instancia (un registro)
 - Para “confirmar” la existencia de una DF es necesario conocer la semántica de sus atributos
 - Para “descartar” la existencia de una DF solo basta mostrar la existencia de tuplas que violan dicha “potencial” DF
- Notación: Conjunto de DF, se denota como F
- Inferencia: diseñador especifica DFs que son semánticamente obvias. Existen otras que se cumplen y que pueden ser inferidas de F.
- Los valores que toman los atributos de Y dependen de los valores que tomen X. Frase. “Y es funcionalmente dependiente de X”
 - Definición 1. Conjunto de atributos X se denominan lado izquierdo de la DF
 - Definición 2. Conjunto de atributos Y se denominan lado derecho de la DF



- La forma normal de una relación refiere a la mayor forma normal alcanzada por ella

Formas normales 1FN

- Prohíbe relaciones dentro de relaciones o relaciones como valores de atributos dentro de tuplas
- Admite el dominio de un atributo debe incluir solo valores atómicos.
- Cada campo debe contener un solo ítem
- Todos los ítems de una columna deben significar lo mismo
- Cada fila debe ser única
- Una tabla no debe tener columnas repetidas

● Ejemplo.

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

● ¿Está en 1FN? ¡No! D_Areas_Influencia no es un atributo atómico

● Técnicas para alcanzar 1FN.

● Técnicas para alcanzar 1FN.

- ① Remover atributo que viola 1FN y ubicarlo en una nueva relación, DEPTO_AREAS, junto con la PK D_Número. La nueva relación tiene como PK ambos atributos

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL
Investigación	2	27-23345876-9
Prensa	3	20-17283948-4
Administración	8	27-38476827-2

DEPTO_AREAS

D_Número	D_Areas_Influencia
2	Argentina
2	Brasil
2	Uruguay
3	Chile
8	Argentina

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Area_Influencia
Investigación	2	27-23345876-9	Argentina
Investigación	2	27-23345876-9	Brasil
Investigación	2	27-23345876-9	Uruguay
Prensa	3	20-17283948-4	Chile
Administración	8	27-38476827-2	Argentina

● ¿Qué problema tiene esta solución? Introduce redundancia en la relación

● Técnicas para alcanzar 1FN.

- ③ Si se conoce la máxima cantidad de valores que puede tomar el atributo, se pueden generar tantos atributos como esa cantidad.

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

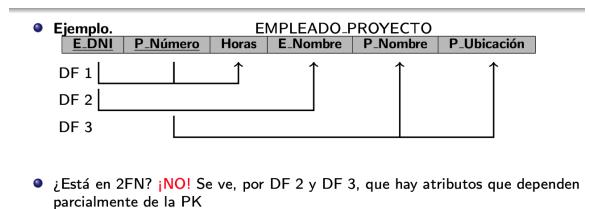
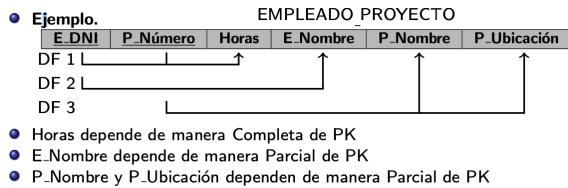
D_NOMBRE	D_Número	D_MGR_CUIL	D_Area_Influencia_1	D_Area_Influencia_2	D_Area_Influencia_3
Investigación	2	27-23345876-9	Uruguay	Brasil	Argentina
Prensa	3	20-17283948-4	Chile	NULL	NULL
Administración	8	27-38476827-2	Argentina	NULL	NULL

● ¿Qué problema tiene esta solución?

- Introducción de valores NULL en casos que la tupla no posee 3 valores para área
- ¿Cuál es la semántica en cuanto a la ubicación de los valores de área?
- Consultas acerca del área se vuelven más complejas. Ej. Listar todos los Departamentos cuya área de influencia incluye a "Argentina"

Formas normales 2FN

- La información debe estar en 1FN
- Las relaciones no deben contener dependencias parciales
- Cada tabla debe ser sobre una sola cosa.
 - Cada campo no clave debe tratarse de lo mismo que la clave primaria
- Las columnas que no son clave, deben depender solamente de la clave completa y no de una parte de la clave.
- DF completa: una $DF \rightarrow Y$ es completa si al eliminar algún atributo A de X la DF deja de existir
- DF parcial: Una $DF \rightarrow Y$ es parcial si es posible eliminar algún atributo A de X y la DF continúa existiendo.
- Un esquema R está en 2FN si todo atributo no primo A de R depende funcionalmente de manera completa de la PK de R
- Verificar sólo DFs cuyos lado izq. posean atributos que sean parte de la PK; si la PK es un solo atributo no es necesario realizar ningún test.



Formas normales 3FN

- Debe estar en 2FN.
- Todos los campos deben depender de la clave completa y nada más que de la clave.
- Una relación no debe tener dependencias funcionales transitivas de la clave primaria
 - No hay otro atributo que no es clave que deba cambiar si cambia un atributo que no es clave.

Una DF $X \rightarrow Y$ en R es Transitiva, si existe un conjunto de atributos Z en R que no son ni Clave Candidata ni un subconjunto de alguna Clave de R, tal que $X \rightarrow Z$ y $Z \rightarrow Y$

Un esquema R está en 3FN si está en 2FN y ningún atributo no primo de R depende transitivamente de la PK



Hay dependencia transitiva porque ni TeacherID ni Teacher name son PK,
pero si cambio uno debo cambiar el otro

Course Title	Fee	Qualification	Teacher ID	Teacher Name
Computer Science	£2000	Advanced Level	1	Miss Lovelace
Mathematics	£2500	Advanced Level	2	Mr Pascal
Physics	£1800	Advanced Level	3	Mr Einstein
Chemistry	£1800	Advanced Level	4	Mr Bunsen
Music	£1200	Diploma	5	Miss Holiday
Biology	£1000	Certificate	6	Mr Darwin
Economics	£1500	Diploma	7	Mr Keynes

Course Title → Teacher ID → Teacher Name

Course Title → Teacher Name → Teacher ID

NoSQL

- Se basa mucho en teorema CAP
- Es una base no-relacional que almacena los datos como documentos estructurados.
- El concepto principal es el documento:
 - La BBDD almacena y recupera documentos
 - Los documentos pueden ser XML, JSON, BSON, etc.
 - Es una colección de pares: nombre de campo y valor. Los valores pueden ser un valor simple o una estructura compleja como listas, otro documento o listas de documentos hijos
- Se basan en esquemas clave:valor. Se usan para bases de grafos, de documentos.
 - Se utiliza una tabla hash en la que una clave única apunta a un elemento
 - Las claves pueden ser organizadas por grupos clave lógicos, requiriendo solamente estas claves para ser únicas dentro de su propio grupo. Esto permite tener claves idénticas en diferentes grupos lógicos.
- Se usa una representación que se llama Document Interaction Diagram (DID)
 - Existen dos formas de graficar, lo que determinará el grado de desnormalización de los documentos:

- Incrustando ($A \rightarrow B$): pongo todo lo que está en A, en B.
- Referenciando ($A \rightarrow B$): A tiene una referencia de B (una key o ID).
- Si un documento se actualiza permanentemente y es muy grande, se pueden crear documentos auxiliares que permitan particionar.
- La BBDD no controla que al referenciar y tener dos documentos que se refieren, sean coherentes en sus ids → No hay integridad referencial.
- Se puede desnormalizar parcialmente (no incrustar todos los key:val, sino algunos).
- Útiles cuando se quiere guardar algo que se repite reiteradas veces y cambia algo.
- La mayoría de las bases de datos disponibles en esta categoría utilizan XML o JSON. Es semi-estructurado en comparación con el RDBMS rígido. Por ejemplo, dos registros pueden tener conjuntos de campos o columnas completamente diferentes. Los registros pueden o no adherirse a un esquema específico (como las definiciones de la tabla en RDBMS).
- Uno de los problemas que soluciona es el escalamiento, porque escala muy bien horizontalmente
 - Escalamiento vertical: agregar cómputo a las computadoras existentes (ram, cpu)
 - Escalamiento horizontal: agregamos nodos o computadoras
- El modelo de datos de Cassandra puede describirse como un store de filas particionadas en donde la información se almacena en tablas hash multidimensionales esparsa.
 - Esparsa significa que para una fila dada, podés tener una o más columnas, pero cada fila no requiere tener todas las mismas columnas que otras filas como en una base relacional.
 - Particionada significa que cada fila tiene una clave única que permite acceder a su info, y las claves se utilizan para distribuir las filas en múltiples stores de datos.

Desnormalización

- Desnormalizar no es una estrategia de diseño, es una solución de diseño.
- Se usa en bases que cumplen paradigma ACID
- Una base de datos correctamente normalizada representa una buena estrategia de diseño, pero puede introducir mucha complejidad de soporte y desarrollo.
 - Para ir a buscar cierta información puede que haya que atravesar muchas tablas para llegar al dato (aunque hay vistas, tablas temp, etc que pueden solucionar esto)
- Idealmente debería hacerse
 1. Durante el diseño inicial
 2. Posterior a la implementación
- Típicamente si el 20% de los stored procedures son usados el 80% del tiempo, se puede considerar desnormalizar.
- Es redituable si ahorra tiempo de desarrollo, tiempo de ejecución
- Lo que ahorra en simplificación de una consulta, podría ser costoso en el mantenimiento futuro
- Normas:
 - Desnormalizar deliberadamente

- Requiere mucho trabajo de documentación porque las desnormalizaciones no suelen ser intuitivas.
- Encapsule las modificaciones de desnormalización en transacciones. Usar la misma lógica de negocio.

El mundo de los sistemas de bases de datos

Database systems: the complete book, by Molina, Ullman, Widom

- El poder de las bases de datos proviene de un conjunto de conocimientos y tecnología que se ha desarrollado durante varias décadas y está incorporado en un software especializado llamado sistema de gestión de bases de datos, o DBMS, o más coloquialmente un “sistema de base de datos”. Un DBMS es una herramienta poderosa para crear y administrar grandes cantidades de datos de manera eficiente y permitiéndoles persistir durante largos períodos de tiempo, sin peligro. Estos sistemas se encuentran entre los tipos de software más complejos disponibles.
- ¿Qué es una base de datos? En esencia no es más que una colección de información que persiste por un largo periodo de tiempo, a menudo muchos años. Es administrado por el DBMS, el cual se espera que:
 - Permita a los usuarios crear tablas y bases de datos, definir sus esquemas (estructura lógica) usando un lenguaje de datos definido (data definition language) / query language.
 - Permitir a los usuarios consultar los datos y modificarlos
 - Soportar el almacenamiento de grandes volúmenes de datos, durante un periodo prolongado, permitiendo su acceso en forma eficiente.
 - Permitir la durabilidad, la recuperación en caso de fallas, errores de distinto tipo o uso indebido.
 - Controlar el acceso a los datos para varios usuarios en simultáneo evitando que interacciones inesperadas entre los mismos (que cada usuario trabaje en aislamiento de los otros) y sin acciones en los datos que se completen en forma parcial y no total (atomicidad).

Bases de datos relacionales

- Ted Codd propuso en 1970 que las bases de datos debían presentar al usuario la información organizada en tablas, llamadas relaciones. Por detrás pueden existir complejas estructuras de datos que permiten respuesta rápida a una variedad de consultas.
- En contraposición a las BBDD anteriores, los programadores de sistemas de bases relationales, no estarían preocupados por la estructura de almacenamiento. Las consultas podrían expresarse en lenguaje de alto nivel, de manera de aumentar la eficiencia.
- Para 1990 las bases relationales eran la norma.

Sistemas más y más pequeños

- Originalmente, los DBMS eran sistemas de software grandes y costosos que se ejecutaban en grandes ordenadores. El tamaño era necesario, porque para almacenar un gigabyte de datos se necesitaba un gran sistema informático. Hoy en día, hay muchos gigabytes en un solo disco y es bastante factible ejecutar un DBMS en una computadora personal. Así, los sistemas basados en el modelo relacional están disponibles incluso para máquinas muy pequeñas y están empezando a aparecer como una herramienta común.

- Otra tendencia es el uso de documentos, a menudo utilizando XML (eXtensible Modeling Language). Grandes colecciones de pequeños documentos pueden servir como BDD y los métodos para consultar y manipularlos son diferentes a las bases de datos relacionales

Integración de la información

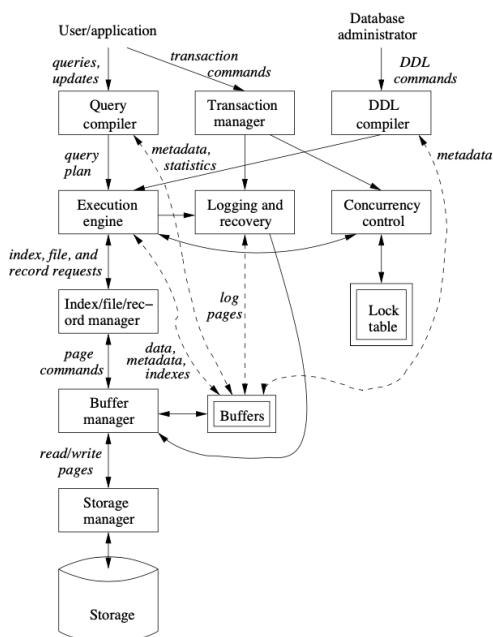
El problema de construir y mantener una BBDD se ha convertido en uno de integrar la información: juntar la información contenida en muchas bases relacionadas, en una sola. Una compañía tiene varias divisiones, cada división quizás construyó su BBDD para los productos, empleados, etc. Quizá incluso eran empresas independientes, usan distintos términos para lo mismo, tienen sistemas legados, con lo cual integrar todo sea hace imposible. Como resultado, se ha hecho popular con frecuencia construir estructuras sobre BBSS existentes con el objetivo de integrar la información contenida entre ellos. Los data warehouses son una solución, en donde muchas bases legacy se copian allí. Otra solución es utilizar middleware, cuya función es soportar un modelo de los datos integrado, traduciendo entre el modelo viejo y el nuevo que utiliza cada base.

Overview de los sistemas de administración de bases de datos DBMS

- Caja simple representa componentes del sistema, caja doble representa estructuras de datos en memoria.
- Flechas sólidas indican flujo de control y de datos, mientras que punteada solo de datos.
- Existen dos fuentes de comandos diferenciadas: usuarios que consultan y modifican datos y administradores de la BBDD que son responsables por la estructura ó esquema de la BBDD.

Comandos del lenguaje de definición de datos

- El administrador de la base de datos, o DBA, de la base de datos de una universidad podría decidir que debe haber una tabla o relación con columnas para un estudiante, un curso que el estudiante ha tomado y una calificación para ese estudiante en ese curso. El administrador de bases de datos también podría decidir que los únicos grados permitidos sean A, B, C, D y F. La información de estructura y restricciones es parte del esquema de la base de datos. El DBA, que necesita autoridad especial ejecutar comandos que alteren el esquema, ya que estos pueden tener efectos profundos en la base de datos. Estos comandos DDL que alteran el esquema (DDL significa data definition language) son analizados por un procesador DDL y pasados al motor de ejecución,



que luego pasa por el administrador de índice/archivo/registro para alterar los metadatos, es decir, la información del esquema de la base de datos.

Descripción general del procesamiento de consultas

- Un usuario o un programa de aplicación inicia alguna acción que no afecta el esquema de la base de datos, pero puede afectar el contenido de la base de datos (si la acción es un comando de modificación) o extraerá datos de la base de datos (si la acción es una consulta). El lenguaje en el que se expresan estos comandos se llama lenguaje de manipulación de datos (DML). Hay lenguajes de manipulación de datos disponibles, pero SQL es el más utilizado. Las declaraciones DML son manejadas por dos subsistemas separados:
 - respondiente la consulta: La consulta es analizada (parseada) y optimizada por un compilador de consultas (query compiler). El plan de ejecución que determinó la DBMS es pasado al motor de ejecución. El motor de ejecución emite una secuencia de solicitudes de pequeños fragmentos de datos, normalmente registros o tuplas de una relación, a un administrador de recursos que conoce los datos (relaciones de retención), el formato y tamaño de los registros, los índices de archivos, que ayudan a encontrar los elementos rápidamente. Los pedidos de información son pasados a un administrador de buffer (buffer manager). Este se encarga de traer las porciones correctas de datos de el almacenamiento secundario (disco) hacia los buffers en memoria. Normalmente, la página o bloque de disco es la unidad de transferencia entre los buffers y el disco. El administrador del buffer se comunica con un administrador de almacenamiento para obtener datos de disco. El administrador de almacenamiento puede involucrar comandos del sistema operativo, pero lo más habitual es que el DBMS emita comandos directamente al controlador de disco.
 - Procesamiento transaccional: Las consultas y otras acciones DML se agrupan en transacciones, que son unidades que deben ejecutarse de forma atómica y aislada unos de otros. La acción de consulta o modificación es una transacción en sí misma. Además, la ejecución de las transacciones debe ser duradera, lo que significa que el efecto de cualquier transacción completada debe preservarse incluso si el sistema falla de alguna manera inmediatamente después de la finalización de la transacción. Dividimos el procesador de transacciones en dos partes principales:
 1. Un administrador o planificador de control de concurrencia, responsable de garantizar atomicidad y aislamiento de transacciones, y
 2. Un administrador de registro y recuperación, responsable de la durabilidad de las transacciones.

Almacenamiento y buffer management

- Los datos de una base de datos normalmente residen en un almacenamiento secundario; En los sistemas informáticos actuales, "almacenamiento secundario" generalmente significa disco magnético. Sin embargo, para realizar cualquier operación útil sobre los datos, esos datos deben

estar en la memoria principal. Es trabajo del administrador de almacenamiento controlar la ubicación de los datos en el disco y su movimiento entre el disco y la memoria principal.

- En un sistema de base de datos simple, el administrador de almacenamiento podría no ser más que que el sistema del sistema operativo subyacente. Sin embargo, por propósitos de eficiencia, los DBMS normalmente controlan el almacenamiento en el disco directamente, al menos bajo algunas circunstancias. El administrador de almacenamiento realiza un seguimiento de su ubicación en el disco y obtiene el bloque o bloques que contienen un archivo a petición del gerente de buffer.
- El administrador de búfer es responsable de dividir la memoria principal disponible en búferes, que son regiones del tamaño de una página a las que se pueden transferir bloques de disco. Por lo tanto, todos los componentes DBMS que necesitan información del disco interactuarán con los buffers y el administrador del buffers, ya sea directamente o a través de el motor de ejecución. Los tipos de información que varios componentes pueden necesitar incluye:
 1. Datos: el contenido de la propia base de datos.
 2. Metadatos: el esquema de la base de datos que describe la estructura y las restricciones de la base de datos.
 3. Registros de logeo: información sobre cambios recientes, estos soportan la durabilidad de la BBDD.
 4. Estadísticas: información recopilada y almacenada por el DBMS sobre los datos. propiedades tales como los tamaños y valores en varias relaciones u otros componentes de la base de datos.
 5. Índices: estructuras de datos que apoyan el acceso eficiente a los datos.

Procesamiento transaccional

- Es normal agrupar una o más operaciones de base de datos en una transacción, que es una unidad de trabajo que debe ejecutarse de forma atómica y en aparente aislamiento de otras transacciones. Además, un DBMS ofrece la garantía de durabilidad: que el trabajo de una transacción completada nunca se perderá. El administrador de transacciones por lo tanto, acepta comandos de transacción de una aplicación, lo que le indica al administrador de transacciones cuándo comienzan y terminan las transacciones, así como información sobre las expectativas de la aplicación (algunos pueden no desear requerir atomicidad, por ejemplo). El procesador de transacciones realiza las siguientes tareas:
 1. Registro/Logging: para garantizar la durabilidad, cada cambio en la base de datos se registra por separado en el disco. El administrador de registros sigue una de varias políticas diseñado para garantizar que, sin importar cuándo ocurra una falla o "caída" del sistema, un administrador de recuperación podrá examinar el registro de cambios y restaurar la base de datos a algún estado consistente. El administrador de registros escribe inicialmente los buffers de inicio de sesión y negocia con el administrador del buffer para asegurarse de que los buffers se escriben en el disco (donde los datos pueden sobrevivir a un fallo) en el momento adecuado.
 2. Control de concurrencia: debe parecer que las transacciones se ejecutan de forma aislada. Pero en la mayoría de los sistemas, en realidad habrá muchas transacciones ejecutándose. Por lo tanto, el planificador (administrador de control de concurrencia) debe garantizar que las acciones individuales de múltiples transacciones se ejecuten de manera tal, que sin importar el orden, el efecto neto sea el mismo que si las transacciones se hubieran realizado en su totalidad, uno a la vez. Un planificador típico lo hace manteniendo bloqueos en ciertas partes de la base de datos. Estos bloqueos evitan que dos transacciones accedan al mismo dato de manera que interactúen

mal. Los bloqueos generalmente se almacenan en una tabla de bloqueos de la memoria principal. El programador afecta la ejecución de consultas y otras operaciones de la base de datos al prohibir que el motor de ejecución acceda a partes bloqueadas de la base de datos.

3. Resolución de bloqueos muertos: a medida que las transacciones compiten por los recursos a través de los bloqueos que otorga el programador, pueden llegar a una situación en la que ninguna puede continuar porque cada una necesita algo que tiene otra transacción. El administrador de transacciones tiene la responsabilidad de intervenir y cancelar (rollback ó abort) una o más transacciones para permitir que las demás continúen.

El procesador de consultas

La parte del DBMS que más afecta el rendimiento que ve el usuario es el procesador de consultas. El procesador de consultas está representado por dos componentes:

1. El compilador de consultas, que traduce la consulta a un formulario interno llamado plan de consulta. Este último es una secuencia de operaciones que se realizarán sobre los datos. A menudo, las operaciones en un plan de consulta son implementaciones de operaciones de "álgebra relacional". El compilador de consultas consta de tres unidades principales:
 - a. Un analizador/parser de consultas, que construye una estructura de árbol a partir del formato textual.
 - b. Un preprocesador de consultas, que realiza comprobaciones semánticas de la consulta (por ejemplo, asegurándose de que todas las relaciones mencionadas en la consulta realmente existan) y realiza algunas transformaciones para convertir el árbol parseado en un árbol de operadores algebraicos que representan el plan de consulta inicial.
 - c. Un optimizador de consultas, que transforma el plan de consulta inicial en el mejor secuencia de operaciones disponible sobre los datos actuales.

El compilador de consultas utiliza metadatos y estadísticas sobre los datos para decidir qué secuencia de operaciones probablemente sea la más rápida. Por ejemplo, la existencia de un índice, que es una estructura de datos especializada que facilita el acceso a los datos, dados valores para uno o más componentes de esos datos, puede hacer que un plan sea mucho más rápido que otro.

2. El motor de ejecución, que tiene la responsabilidad de ejecutar cada uno de los pasos del plan de consulta elegido. El motor de ejecución interactúa con la mayoría de los otros componentes del DBMS, ya sea directamente o a través de los amortiguadores. Debe obtener los datos de la base de datos en buffers para poder manipular esos datos. Necesita interactuar con el planificador para evitar acceder a datos que están bloqueados y con el administrador de registros para asegurarse de que todos los cambios en la base de datos se registran correctamente.

Bases de datos de grafos

Nombre completo: "Base de datos de grafos con propiedades de etiquetas"

- Es una base de datos, no es una visualización de datos.
- Paradigma ACID (no BASIC). Es un base de datos transaccional, es decir, que cumple con las características ACID.
- Así como las bases de datos relacionales guardan los datos en forma de tabla, Neo4j los guarda en forma de grafos, como un network, una red.

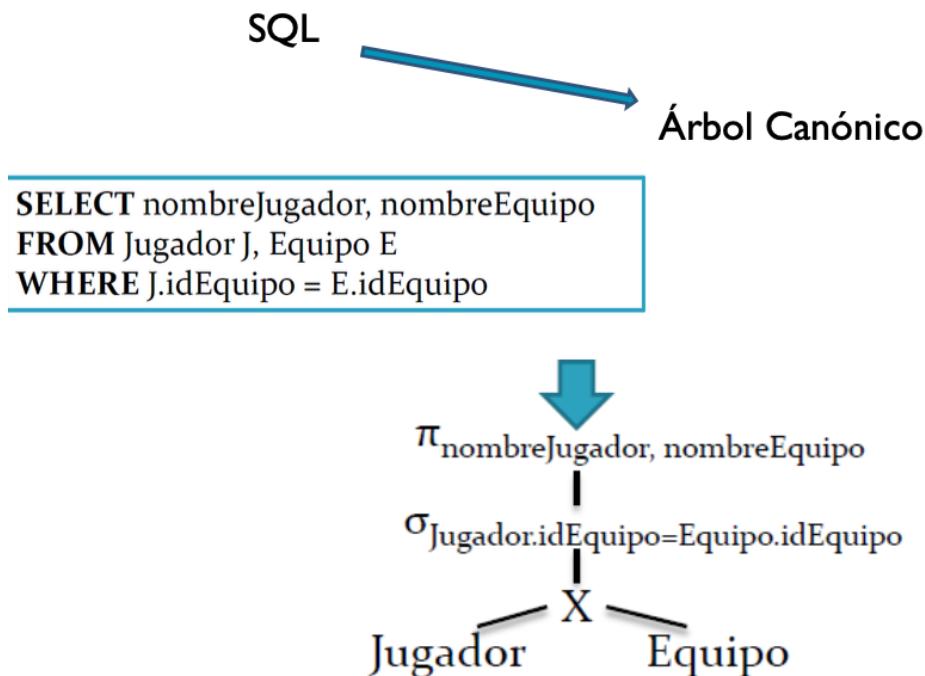
- Relaciones: Gráficamente corresponden a las aristas o líneas que unen los vértices. Para Neo4j esta característica genera una relación entre los puntos que conecta.
- Nodos: Son los vértices que unen las relaciones.
- Camino: conjunto de vértices interconectados por aristas.
- Propiedades: son características adicionales que se le pueden asignar tanto a los nodos como a las relaciones para otorgar información adicional y enriquecer el modelo.
- Un nodo típicamente representa una entidad, como una persona, producto o diagnóstico de un paciente. Opcionalmente se le pueden agregar etiquetas a un nodo, indicando el rol del nodo dentro del grafo.
- Las relaciones: Para unir nodos se utilizan relaciones. Las relaciones son dirigidas, y pueden eventualmente tener propiedades asociadas. El tipo de relación provee de un predicado, mientras que la dirección de una relación muestra el sujeto y objeto.
- Etiquetas: los grafos llevan incorporadas etiquetas que pueden definir los distintos vértices y también las relaciones entre ellos. Por ejemplo, en Facebook podríamos tener nodos definidos por términos como 'amigo' o 'compañero de trabajo' y la relaciones como 'amigo de' o 'socio de'. Con etiquetas podemos asignar propiedades tanto a nodos como relaciones (por ejemplo, cuestiones como nombre, edad, país de residencia, nacimiento).
- **Cypher query** sintaxis (o SQL for graphs): es un lenguaje declarativo. Qué es lo que queremos, no cómo acceder a los datos.
 - Cada nodo puede tener propiedades
 - Los nodos pueden ser etiquetados con una o más etiquetas
 - Las relaciones también pueden tener nombre y propiedades

Optimización

- Es importante que la BBDD tenga actualizado el catálogo (`analyze verbose tb`). Hay un comando más exhaustivo que puede hacerse semanalmente que recoge estadísticas, dtypes, etc que luego servirá para que el motor optimice queries. (`collect statistics`)
- Una BD se almacena como una colección de archivos. Cada archivo contiene registros del mismo tipo y se divide en bloques de igual tamaño.
- Organización de Archivos se refiere a la forma en que los datos son almacenados dentro de un archivo y las formas en que pueden accederse. Dependiendo de cómo están guardados los datos cambia la complejidad algorítmica.
 - HeapFile: no ordenado, se inserta secuencial al final del archivo o en alguno de los bloques con espacio libre. Las operaciones de búsqueda requieren búsqueda lineal por todos los bloques del archivo. Barato para insertar, caro para buscar y/o modificar
 - SortedFile: registros ordenados a partir de una clave de búsqueda. Al insertarse un registro se lo agrega ordenadamente, lo que puede provocar una reorganización en los bloques del archivo. Mejoran las búsquedas por la clave, pero el resto de las operaciones suelen requerir una búsqueda lineal. Barato para buscar, caro para insertar porque requiere reordenar con cada inserción.

- Estructuras adicionales aceleran ciertas operaciones de búsqueda sobre tablas. Las dos estructuras clásicas son árboles balanceados B+ y tablas Hash.
 - Mayor costo en operaciones de escritura, actualización y borrado
 - Mayor costo en espacio ocupado
- Hay un índice primario (único) general, ordenado.
- Índice secundario: tiene otra estructura y gasto de procesamiento, puede haber varios.
- Una tabla de hash almacena las claves de búsqueda. Armo buckets en donde la función de hash es el que asigna al bucket. Luego en el bucket busco linealmente. Cada posición de una tabla hash se asocia con un conjunto de registros. Por esta razón cada posición suele llamarse “un bucket”, y los valores de hash, “índices bucket”. Son buenos si la query involucra comparaciones de igualdad.
- Cuando un usuario formula una consulta, se analiza y envía esta consulta a un optimizador de consultas, que utiliza información sobre el modo que se guardan los datos para producir un plan de ejecución eficiente para la evaluación de esa consulta. Un plan de ejecución es un detalle de las acciones que debe realizar el motor para la evaluación de la consulta, representado habitualmente como un árbol de operadores con anotaciones que contiene información detallada adicional sobre los métodos de acceso que se deben emplear, etc.
- El plan de ejecución no puede analizar y buscar el mejor plan, pero puede encontrar uno eficiente.
 1. Arma el árbol canónico
 2. Buscar optimizarlo para mejorar la performance de la consulta independientemente de la organización física. Involucran propiedades que permiten construir una consulta equivalente a la original.
- Técnicas algebraicas. Heurísticas Árboles equivalentes. Por lo general, mejoran la performance de las consultas
- Técnicas físicas: define un modelo de costos read/write. En general predomina sobre CPU. Los costos se estiman y se elige el de menor costo.
 - Seleccionar implementaciones para los operadores basándose en cómo están organizados los archivos y las estructuras adicionales que existen. Utilizan el Catálogo de Información estadística de los datos. Se actualiza periódicamente y no está siempre sincronizado con los datos reales. Permite estimar la selectividad de los diferentes operadores.
 - No se asume nada del buffer manager, siempre se considera que un pedido de R/W implica acceder a disco.
 - En un bloque de R, hay solo tuplas de R.
 - Las tuplas de R se guardan enteras en un bloque.
 - Siempre se asume el peor caso.
 - Existen dos formas de trabajo para pasar resultados entre nodos:
 - Materialización: Los resultados intermedios se guardan directamente en disco. El siguiente paso los levantará.
 - Pipeline: Las tuplas se van pasando al nodo superior del arbol mientras se continúa ejecutando la operación

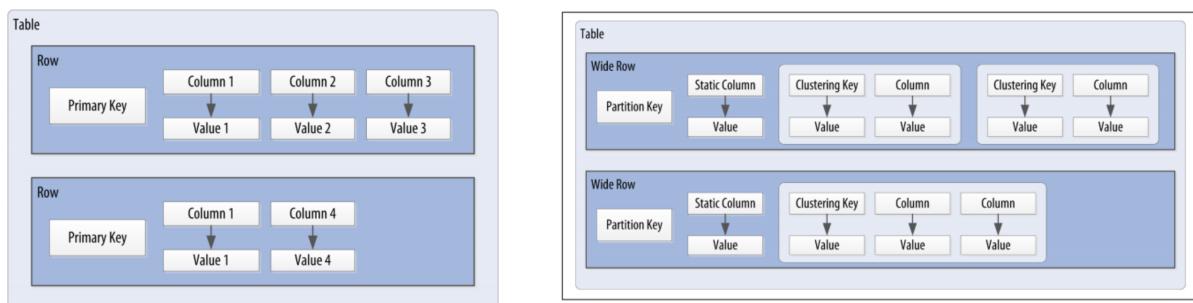
- Block Nested Loops Join (BNLJ): no requiere estructuras extras, pego todo lo que le pido. un algoritmo de junta que consiste en comparar una a una las condiciones del join sin usar estructuras adicionales
- Index Nested Loops Join: usa un índice para pegar. Mira uno por uno
- Sort Merge Join (SMJ).
- Hash join



Wide column (Cassandra)

- El modelo de datos de cassandra puede describirse como un almacen de filas particionado, donde los datos se almacenan en tablas hash esparsas y multidimensionales.
 - Esparsa significa que para una fila dada, podés tener una o más columnas, pero cada fila no requiere tener todas las mismas columnas que otras filas como en una base relacional.
 - Particionada significa que cada fila tiene una clave única que permite acceder a su info, y las claves se utilizan para distribuir las filas en múltiples stores de datos.
- Cassandra define una tabla como una división lógica que asocia información similar
- Los campos de cada tabla son opcionales, no todos tienen que tener los mismos, ni la misma cantidad. Se evitan de esta manera nulls.
- Cuando se diseñan las filas, se puede hacer que sean anchas o flacas. Una fila ancha significa que tiene potencialmente miles o millones de columnas
- Cassandra requerirá que restrinja todas las columnas de clave de partición o ninguna de ellas a menos que la consulta pueda usar un índice secundario..
- Modelo Conceptual (DER) y Workflow (consultas)

- Se crea una tabla por consulta
- Modelo Lógico
 - Buscar el subconjunto del modelo conceptual que satisface cada consulta
 - Elegir claves
 - Usar diagramas Chebotko para describir el modelo lógico
- Reglas de Mapeo: Basados en los DMP (Data Modeling Principles), las reglas de mapeo ayudan a realizar la transición desde el modelo conceptual al modelo lógico.
- **MR1** (Entities and Relationships): Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas
- **MR2** (Equality Search Attributes): Si se utilizan en una consulta por igualdad de atributos, entonces, éstos se mapean a columnas del prefijo de la clave primaria. Dichas columnas deben incluir todas las columnas de clave de partición y, opcionalmente, una o más columnas clustering key.
- **MR3** (Inequality Search Attributes): Si se utilizan en consultas por desigualdad, estos atributos mapean como columnas clustering key. En la definición de clave principal, una columna que participa en la búsqueda de desigualdad debe ubicarse después de las columnas que participan en la búsqueda de igualdad.
- **MR4** (Ordering Attributes): Mapea a una columna clustering key con orden ascendente o descendente según se especifique en la consulta
- **MR5**(Key Attributes): Mapea a columnas en la clave primaria. Una tabla que almacena datos de entidades o relaciones como filas debe incluir atributos claves que identifique estos datos únicamente



Control de concurrencia

- Cuando una aplicación manda un request de un pedido transaccional tiene una estructura `start, read, write, commit`. El `commit` significa que se confirma la transacción y no se puede deshacer.
- Cuando el motor scheduler recibe en concurrencia distintos pedidos hay dos políticas posibles optimista y pesimista.
- Por ACID debería suceder que si ordeno por pedido, todo quede igual si los ejecuto desordenadamente.
- Asumiendo que la trx 1 arrancó primero:

- Read too late: Si la trx 2 escribe y luego la 1 lee, no estoy respetando ACID porque el resultado final no será igual a si lo hubiera ejecutado como llegó. Se tiene que abortar
 - Write too late: La trx 1 debería escribir antes de leer.
 - Dirty data: sucede cuando la trx 1 tiene un abort pero la trx 2 lee previo al abort de la 1.
 - Regla de Thomas: cuando hay 2 trx que escriben y suceden en orden inverso.
- Si una trx es sancionada se hace rollback

O'REILLY®

2nd Edition



Cassandra

The Definitive Guide

DISTRIBUTED DATA AT WEB SCALE

Jeff Carpenter & Eben Hewitt

Row-Oriented

Cassandra's data model can be described as a partitioned row store, in which data is stored in sparse multidimensional hashtables. "Sparse" means that for any given row you can have one or more columns, but each row doesn't need to have all the same columns as other rows like it (as in a relational model). "Partitioned" means that each

row has a unique key which makes its data accessible, and the keys are used to distribute the rows across multiple data stores.

Cassandra's Data Model

In this section, we'll take a bottom-up approach to understanding Cassandra's data model.

The simplest data store you would conceivably want to work with might be an array or list. It would look like [Figure 4-1](#).



Figure 4-1. A list of values

So we'd like to add a second dimension to this list: names to match the values. We'll give names to each cell, and now we have a map structure, as shown in [Figure 4-2](#).

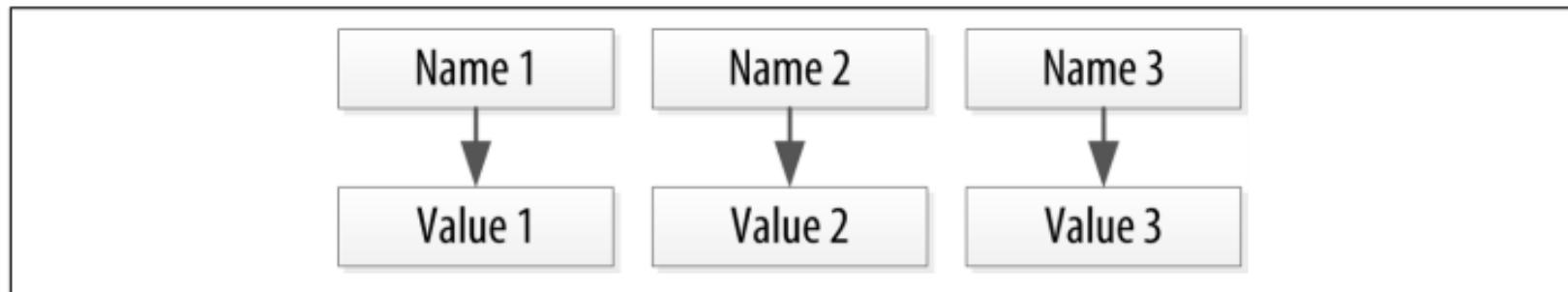


Figure 4-2. A map of name/value pairs

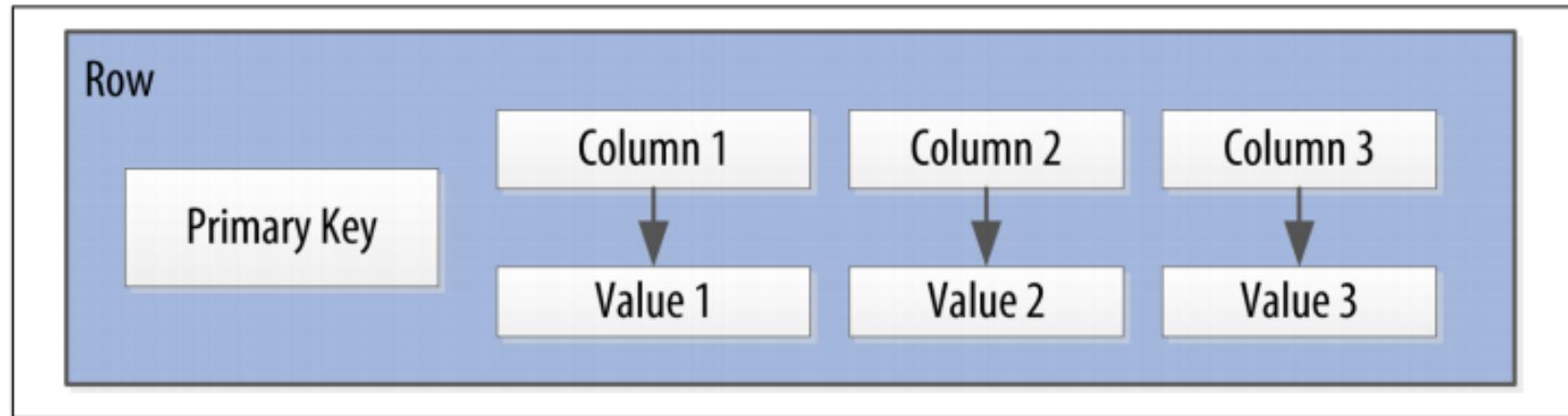


Figure 4-3. A Cassandra row

Cassandra defines a *table* to be a logical division that associates similar data. For example, we might have a user table, a hotel table, an address book table, and so on. In this way, a Cassandra table is analogous to a table in the relational world.

Now we don't need to store a value for every column every time we store a new entity. Maybe we don't know the values for every column for a given entity. For example, some people have a second phone number and some don't, and in an online form backed by Cassandra, there may be some fields that are optional and some that are required. That's OK. Instead of storing null for those values we don't know, which would waste space, we just won't store that column at all for that row. So now we have a sparse, multidimensional array structure that looks like Figure 4-4.

Table

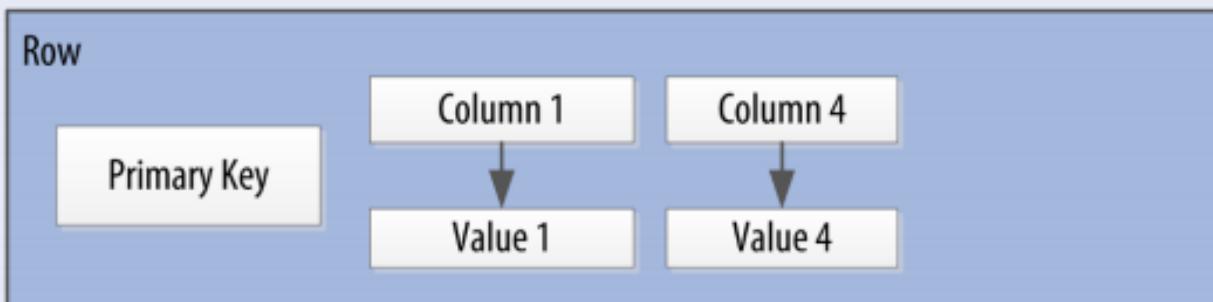
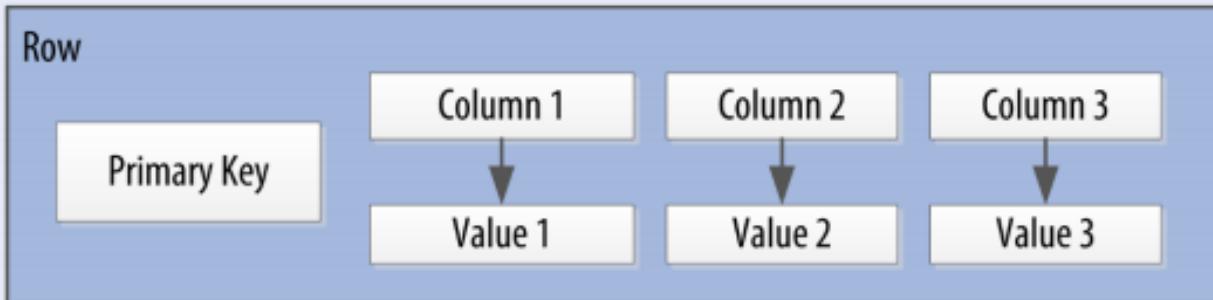


Figure 4-4. A Cassandra table

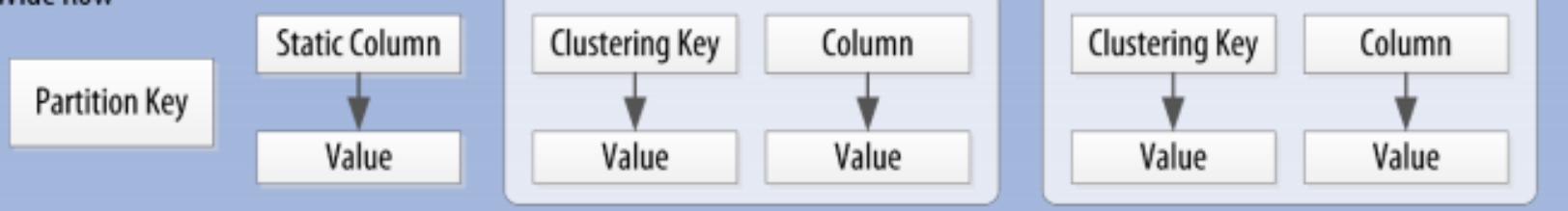
	Column 1	Column 2	Column 3	Column 4
Row 1	Value 1	Value 2	Value 3	
Row 2	Value 1			Value 4

When designing a table in a traditional relational database, you're typically dealing with "entities," or the set of attributes that describe a particular noun (`hotel`, `user`, `product`, etc.). Not much thought is given to the size of the rows themselves, because row size isn't negotiable once you've decided what noun your table represents. However, when you're working with Cassandra, you actually have a decision to make about the size of your rows: they can be wide or skinny, depending on the number of columns the row contains.

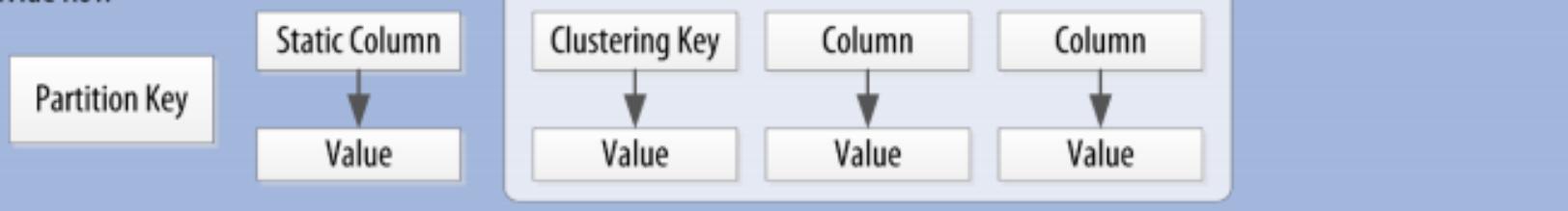
A wide row means a row that has lots and lots (perhaps tens of thousands or even millions) of columns. Typically there is a smaller number of rows that go along with so many columns. Conversely, you could have something closer to a relational model, where you define a smaller number of columns and use many different rows—that's the skinny model. We've already seen a skinny model in [Figure 4-4](#).

Table

Wide Row



Wide Row



PRIMARY KEY: ((*cust_id*), *event_time*)

customer_events table

partition key

cust_id **K**
↓
kTe82rn2

cust_name **S**
↓
"Fred Flinstone"

cust_email **S**
↓
"fred@gmail.com"

multiple event records

event_time **C**
↓
2018-06-02T19:42:31

event_type
↓
"web login"

event_time **C**
↓
2018-06-02T19:43:55

event_type
↓
"purchase"

...

cust_id **K**
↓
rTq59vd6

cust_name **S**
↓
"Joe Rockhead"

cust_email **S**
↓
"joer@yahoo.com"

event_time **C**
↓
2018-06-02T19:42:38

event_type
↓
"refund"

event_time **C**
↓
2019-03-11T06:24:03

event_type
↓
"web login"

...

cust_id **K**
↓
N6pDgQ5G

cust_name **S**
↓
"Wilma Flinstone"

cust_email **S**
↓
"wilma@bedrock.org"

event_time **C**
↓
2018-06-02T19:43:02

event_type
↓
"help line"

agent_name
↓
"Phillips"

...
K – partition key
C – clustering key
S – static columns

Introducción

HBase
ooooo

Cassandra
oooooooooooooooooooo

Un poco de CQL
●○○○○○○○

Diseño

Un poco de CQL

Tipos de Datos CQL

- Numéricos: bigint, smallint, tinyint, varint, float, double, decimal
 - Texto: text, varchar (UTF-8 character string), ascii
 - Tiempo: timestamp, date, time
 - Identificadores: uuid, timeuuid
 - Otros: boolean, blob, inet, **counter**
 - Colecciones: set, list y map.
 - Tuple. Ejemplo: address tuple<text, text, text, int>;
 - User-Defined Types.

Crear una Tabla

```
CREATE TABLE IF NOT EXISTS rank_by_year_and_name (  
race_year int,  
race_name text,  
cyclist_name text,  
rank int,  
PRIMARY KEY ((race_year, race_name), rank));
```

```
INSERT INTO cycling.rank_by_year_and_name  
(race_year, race_name, cyclist_name, rank)  
VALUES (2015,  
'Tour of Japan - Stage 4 - Minami > Shinshu',  
'Benjamin PRADES', 1);
```

Restricción Clave de Partición

Operadores - Clave de Partición

Las columnas de la clave de partición solo admiten dos operadores: = e IN

Cassandra requerirá que restrinja todas las columnas de clave de partición o ninguna de ellas a menos que la consulta pueda usar un índice secundario.

Where en Clave de Partición

```
SELECT rank, cyclist_name as name  
FROM cycling.rank_by_year_and_name  
WHERE race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'  
AND race_year = 2015  
AND rank <= 2;
```

Where - Cluster Columns

```
CREATE TABLE numberOfRequests
(cluster text,
date text,
datacenter text,
hour int,
minute int,
numberOfRequests int,
PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

Where - Cluster Columns

CORRECTO:

```
SELECT * FROM numberofRequests  
WHERE cluster = 'cluster1' AND date = '2015-06-05'  
AND datacenter = 'US_WEST_COAST' AND hour = 14 AND minute = 00;
```

INCORRECTO:

```
SELECT * FROM numberOfRequests  
WHERE cluster = 'cluster1' AND date = '2015-06-05'  
AND hour = 14 AND minute = 00;
```

Where - Cluster Columns

Correctas:

```
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND
date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND
hour= 12 AND minute >= 0 AND minute <= 30;
```

```
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND
    date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND
    hour >= 12;
```

```
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND
    date = '2015-06-05' AND datacenter > 'US';
```

Incorrecta:

```
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour >= 12 AND minute = 0;
```

Introducción

HBase
ooooo

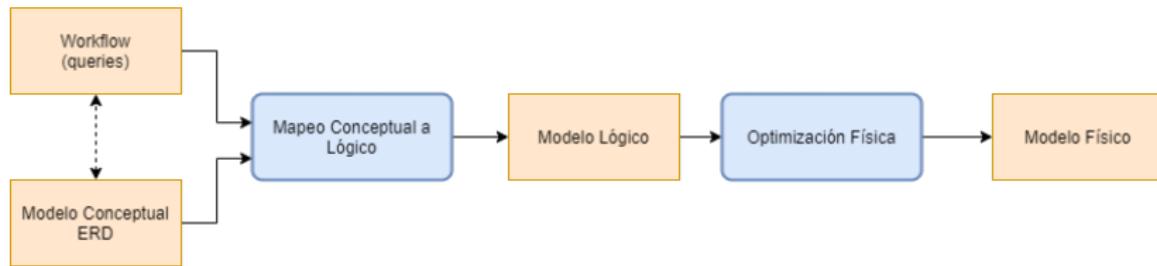
Cassandra
oooooooooooooooooooo

Un poco de CQL
oooooooo

Diseño
●oooooooooooooooooooo

Diseño

Cassandra - Notación/Método Chebotko



- Modelo Conceptual (DER) y Workflow (consultas)
 - Modelo Lógico
 - Buscar el subconjunto del modelo conceptual que satisface cada consulta
 - Elegir claves
 - Usar diagramas Chebotko para describir el modelo lógico

SE CREA UNA TABLA POR CONSULTA

Reglas de Mapeo

Basados en los DMP (Data Modeling Principles), las reglas de mapeo ayudan a realizar la transición desde el modelo conceptual al modelo lógico.

- 1** **MR1 (Entities and Relationships):** Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas

Reglas de Mapeo

Basados en los DMP (Data Modeling Principles), las reglas de mapeo ayudan a realizar la transición desde el modelo conceptual al modelo lógico.

- 1 **MR1 (Entities and Relationships)**: Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas
 - 2 **MR2 (Equality Search Attributes)**: Si se utilizan en una consulta por igualdad de atributos, entonces, éstos se mapean a columnas del prefijo de la clave primaria. Dichas columnas deben incluir todas las columnas de clave de partición y, opcionalmente, una o más columnas clustering key.

Reglas de Mapeo

Basados en los DMP (Data Modeling Principles), las reglas de mapeo ayudan a realizar la transición desde el modelo conceptual al modelo lógico.

- 1 **MR1 (Entities and Relationships)**: Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas
 - 2 **MR2 (Equality Search Attributes)**: Si se utilizan en una consulta por igualdad de atributos, entonces, éstos se mapean a columnas del prefijo de la clave primaria. Dichas columnas deben incluir todas las columnas de clave de partición y, opcionalmente, una o más columnas clustering key.
 - 3 **MR3 (Inequality Search Attributes)**: Si se utilizan en consultas por desigualdad, estos atributos mapean como columnas clustering key. En la definición de clave principal, una columna que participa en la búsqueda de desigualdad debe ubicarse después de las columnas que participan en la búsqueda de igualdad.

Reglas de mapeo. Continuación

- ④ **MR4 (Ordering Attributes)**: Mapea a una columna clustering key con orden ascendente o descendente según se especifique en la consulta
 - ⑤ **MR5 (Key Attributes)**: Mapea a columnas en la clave primaria. Una tabla que almacena datos de entidades o relaciones como filas debe incluir atributos claves que identifique estos datos únicamente

Cassandra - Notación/Método Chebotko

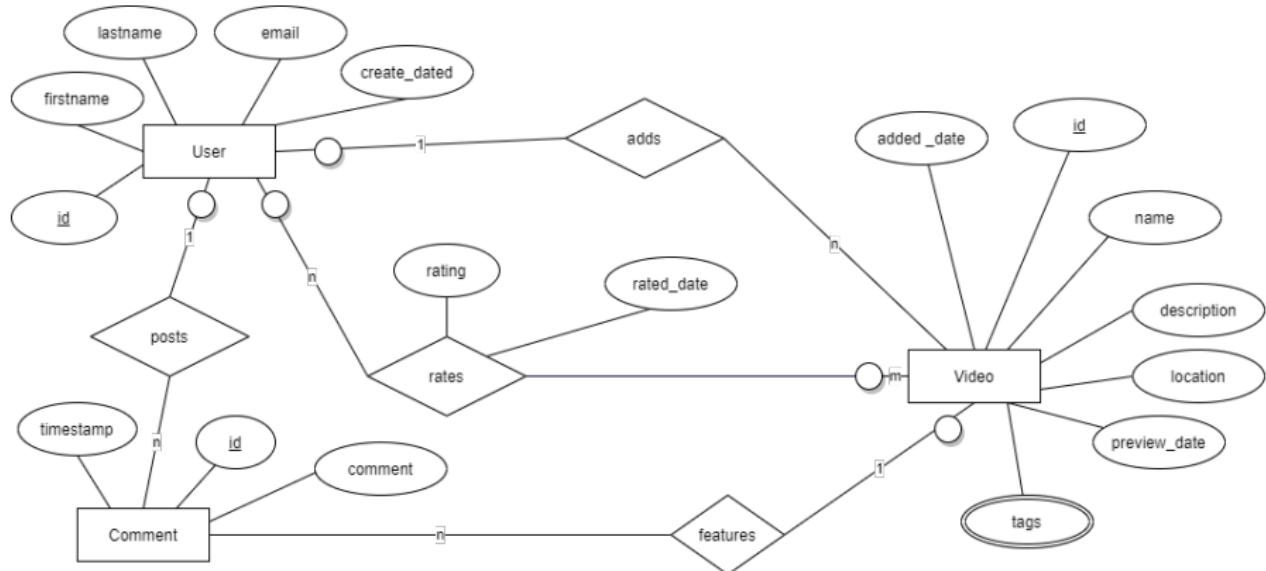
table_name			
column_name_1	CQL Type	K	Partition key column
column_name_2	CQL Type	CT	Clustering key column (ASC)
column_name_3	CQL Type	CD	Clustering key column (DESC)
column_name_4	CQL Type	S	Static column
column_name_5	CQL Type	IDX	Secondary index column
column_name_6	CQL Type	++	Counter column
[column_name_7]	CQL Type		Collection column (list)
{column_name_8}	CQL Type		Collection column (set)
<column_name_9>	CQL Type		Collection column (map)
column_name_10	UDT Name		UDT column
(column_name_11)	CQL Type		Tuple column
column_name_12	CQL Type		Regular column

Ejemplo

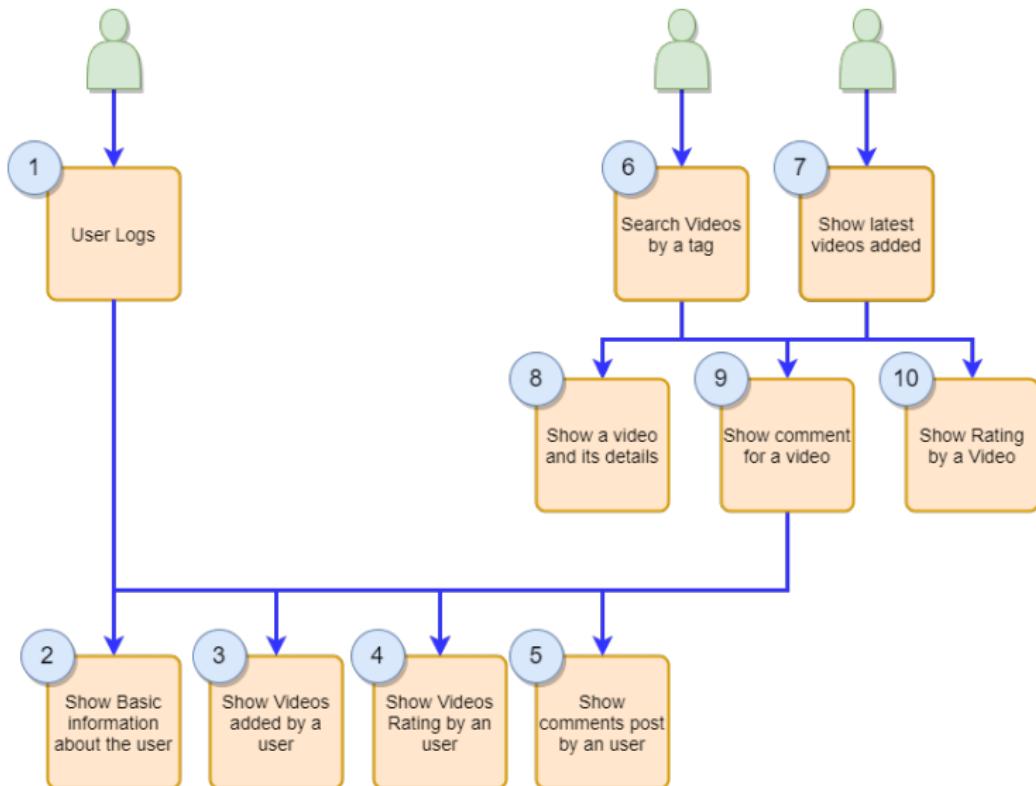
Aplicación de videos, los usuarios suben videos, califican los videos y comentan videos

Ejemplo

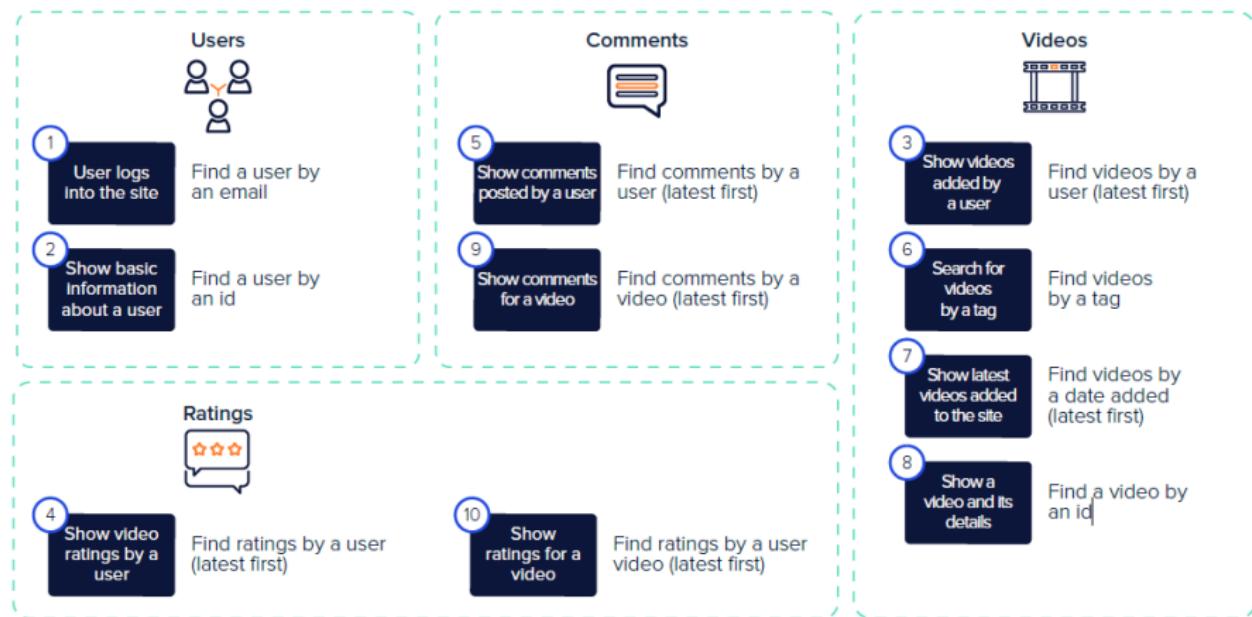
Aplicación de videos, los usuarios suben videos, califican los videos y comentan videos



Example Workflow



Identificar queries



Queries

- 1 user_by_email
 - 2 user_by_id
 - 3 videos_by_user
 - 4 ratings_by_user
 - 5 comments_by_user
 - 6 videos_by_tag
 - 7 latest_videos
 - 8 videos_by_id
 - 9 comments_by_video
 - 10 ratings_by_video

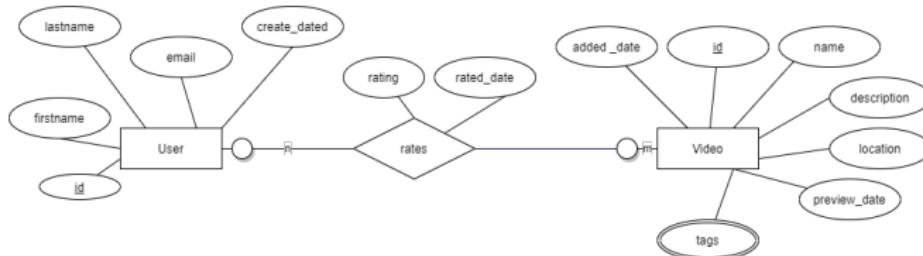
Tablas x Query

1-User by email / 2-User by Id

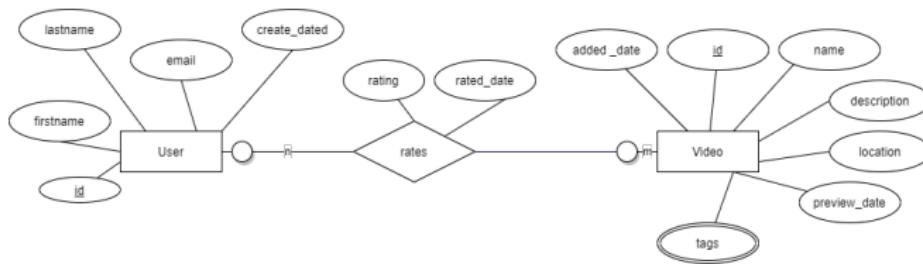
user_by_email	
email	K
userid	C↑
password	

users	
userid	K
firstname	S
lastname	S
email	S
created_date	S

10-Ratings by Video con rated_date descendente y video info



10-Ratings by Video con rated_date descendente y video info



ratings_by_video	
videoid	K
rated_date	C↓
userid	C↑
rating	
added_date	S
location	S
descripcion	S
preview_date	S
{tags}	S

Modelo Físico

ratings_by_video	
videoid	K
rated_date	C↓
userid	C↑
rating	
added_date	S
location	S
descripcion	S
preview_date	S
{tags}	S



Modelo Físico

ratings_by_video	
videoid	K
rated_date	C↓
userid	C↑
rating	
added_date	S
location	S
descripcion	S
preview_date	S
{tags}	S



ratings_by_video		
videoid	uuid	K
rated_date	timestamp	C↓
userid	uuid	C↑
rating	int	
added_date	date	S
location	text	S
descripcion	text	S
preview_date	text	S
{tags}	set<text>	S

CQL

```
CREATE TABLE raiting_by_video(
    videoid uuid,
    rated_date,
    userid uuid,
    rating int ,
    added date,
    location text,
    descripcion text,
    preview_date text,
    tags set<text> ,
    PRIMARY KEY((videoid), rated_date, userid)
) WITH CLUSTERING ORDER BY (rated_date DESC, userid ASC)
```

Bibliografía

- Artem Chebotko, Andrey Kashlev, and Shiyong Lu. 2015. *A Big Data Modeling Methodology for Apache Cassandra*. In Proceedings of the 2015 IEEE International Congress on Big Data.
 - Jeff Carpenter and Eben Hewitt. 2020. *Cassandra: The Definitive Guide* (3rd. ed.). O'Reilly Media, Inc.

Normalización

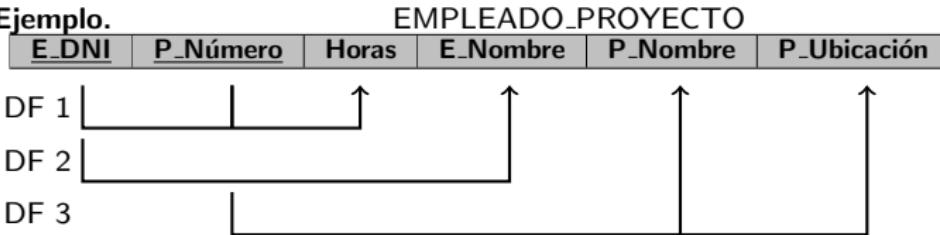
2023



Dependencias Funcionales

- **Propósito.** Herramienta formal para el análisis de esquemas. Permite detectar y describir problemas descriptos previamente
- **Informalmente.** Restricción entre dos conjuntos de atributos X e Y de una BD. Los valores que toman los atributos de Y dependen de los valores que tomen X

- **Ejemplo.**

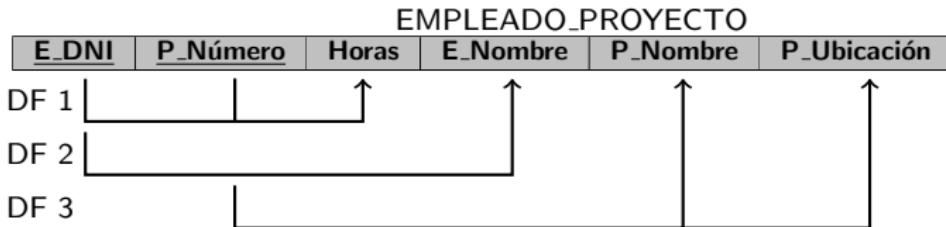


- **DFs.**

- $\{E_DNI, P_Número\} \rightarrow Horas$
- $E_DNI \rightarrow E_Nombre$
- $P_Número \rightarrow \{P_Nombre, P_Ubicación\}$

Dependencias Funcionales

- **Ejemplo.**



- **DFs.**

- $\{E_DNI, P_Número\} \rightarrow Horas$
- $E_DNI \rightarrow E_Nombre$
- $P_Número \rightarrow \{P_Nombre, P_Ubicación\}$

- **Frase.** "Y es funcionalmente dependiente de X"

- **Definición 1.** Conjunto de atributos X se denominan **lado izquierdo** de la DF

- **Definición 2.** Conjunto de atributos Y se denominan **lado derecho** de la DF

Dependencias Funcionales

- **Semántica.** DF son propiedad de la semántica (o significado) de los atributos.
- **Diseño.** Diseñadores de las BD deben usar su entendimiento de la semántica de atributos de R para especificar las DF y deberán respetar TODOS los $r(R)$
- **Instancias legales.** registro que satisface restricciones de DF se denomina instancia legal, estado legal
- **Inferencia de DF.** Dada una relación con sus datos, no es posible determinar sus DF a través de sus valores. Es necesario conocer el significado y relación que existe entre los atributos que la componen
- **Ejemplo. ¿Cuáles son las DF?**

IDMascota	Nombre	Vacuna	Fecha
1	Homy	Antirrábica	26-04-2019
2	Panda	Triple	13-02-2020
2	Panda	Antirrábica	22-01-202020
1	Homy	Antirrábica	26-04-2018

- **Existencia.** Una DF puede existir si la cumple una instancia (un registro)
 - Para "confirmar" la existencia de una DF es necesario conocer la semántica de sus atributos
 - Para "descartar" la existencia de una DF sólo basta mostrar la existencia de tuplas que violan dicha "potencial" DF
- **Notación.** Conjunto de DF, se denota como F
- **Inferencia.** Diseñador especifica DFs que son semánticamente obvias. Existen otras que se cumplen y que pueden ser inferidas de F .

FN basadas en PK

- **Se asume.**
 - Se cuenta con el conjunto de DF para cada relación
 - Cada relación tiene designada su Clave Primaria (PK)
- **Proceso de Normalización.**
 - Propuesto por Codd (1972a)
 - A cada esquema ejecutarle una serie de test para **certificar** que satisface una **forma normal**
- **Normalización de los datos.**
 - Proceso de analizar los esquemas, basándose en DF y PK
 - Objetivo: lograr propiedades deseables
 - Minimizar redundancia
 - Minimizar anomalías de inserción, delección y modificación
 - Esquemas que no pasan ciertos test de **formas normales**, se decomponen en esquemas más pequeños que pasan el test (y sus propiedades)
- **Definición.** La **forma normal** de una relación refiere a la mayor forma normal alcanzada por ella

FN basadas en PK

- **Sin garantía.** Las formas normales, consideradas aisladas de otros factores, no garantizan un buen diseño de la BD
- **Propiedades.** Luego de proceso de normalización por descomposición
 - **Nonadditive Join (Lossless Join).** Garantía de que no ocurre problema de generación de tuplas espúreas. La relación original tiene que poder ser recuperada de la descomposición.
 - **Preservación de DF.** Garantía de que cada DF se encuentra representada en algún esquema resultante de la descomposición
- **Lossless Join** debe lograrse a cualquier costo
- **Preservación de DF.** Es deseable, pero en algunos casos es sacrificada

FN basadas en PK - 1FN

- **1FN.**

- **Prohibe** relaciones dentro de relaciones o relaciones como valores de atributos dentro de tuplas
- **Admite** El dominio de un atributo debe incluir sólo valores atómicos (simples e indivisibles). En la tupla, puede tomar 1 solo valor del dominio.

- **Ejemplo.**

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

- ¿Está en 1FN? ¡No! D_Areas_Influencia no es un atributo atómico

FN basadas en PK - 1FN

- Técnicas para alcanzar 1FN.

- ① Remover atributo que viola 1FN y ubicarlo en una nueva relación, DEPTO_AREAS, junto con la PK D_Número. La nueva relación tiene como PK ambos atributos

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL
Investigación	2	27-23345876-9
Prensa	3	20-17283948-4
Administración	8	27-38476827-2

DEPTO_AREAS

D_Número	D_Areas_Influencia
2	Argentina
2	Brasil
2	Uruguay
3	Chile
8	Argentina

FN basadas en PK - 1FN

- Técnicas para alcanzar 1FN.
 - 2 Expandir la PK que permita que exista más de un mismo D_Número, pero con distinta área de influencia.

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Area_Influencia
Investigación	2	27-23345876-9	Argentina
Investigación	2	27-23345876-9	Brasil
Investigación	2	27-23345876-9	Uruguay
Prensa	3	20-17283948-4	Chile
Administración	8	27-38476827-2	Argentina

- ¿Qué problema tiene esta solución? Introduce *redundancia* en la relación

FN basadas en PK - 1FN

- **Técnicas para alcanzar 1FN.**

- ③ Si se conoce la máxima cantidad de valores que puede tomar el atributo, se pueden generar tantos atributos como esa cantidad.

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Areas_Influencia
Investigación	2	27-23345876-9	{Argentina, Brasil, Uruguay}
Prensa	3	20-17283948-4	{Chile}
Administración	8	27-38476827-2	{Argentina}

DEPARTAMENTO

D_NOMBRE	D_Número	D_MGR_CUIL	D_Area_Influencia_1	D_Area_Influencia_2	D_Area_Influencia_3
Investigación	2	27-23345876-9	Uruguay	Brasil	Argentina
Prensa	3	20-17283948-4	Chile	NULL	NULL
Administración	8	27-38476827-2	Argentina	NULL	NULL

- ¿Qué problema tiene esta solución?

- Introducción de valores NULL en casos que la tupla no posee 3 valores para área
- ¿Cuál es la semántica en cuanto a la ubicación de los valores de área?
- Consultas acerca del área se vuelven más complejas. Ej. Listar todos los Departamentos cuya área de influencia incluye a "Argentina"

FN basadas en PK - 1FN

- **Mejor solución.** La primer opción suele ser la mejor porque no sufre de redundancia y es genérica (no se limita a un máximo de valores posibles): Remover atributo que viola 1FN y ubicarlo en una nueva relación, junto con la PK.
- **Recursividad.** La Técnica se puede utilizar recursivamente para múltiples niveles
- **Múltiples atributos multivaluados.** Debe manejarse con cuidado
- **Ejemplo.**

PERSONA

P_CUIL	P_Cédula_Azul	Teléfonos
27-23345876-9	{JYF 456, PFR 345, KOL 102}	{11-4567-2321, 11-6783-9283}
20-17283948-4	{RUI 234, FGH 736}	{2345-423-3456, 11-2343-2342, 11-2321-2321}

- Aplicando “textualmente” Estrategia Nro. 2.

PERSONA_CEDULA_TELÉFONO

P_CUIL | P_Cédula_Azul | P_Teléfono

- ¿Qué problema produce? **Genera relación no existente entre P_Cédula_Azul y P_Teléfono**
- **Solución.** Utilizar Estrategia Nro. 1

PERSONA_CÉDULA

P_CUIL | P_Cédula_Azul

PERSONA_TELÉFONO

P_CUIL | P_Teléfonos

FN basadas en PK - 1FN

- **Relaciones anidadas.** Cuando el valor de una tupla es una relación.
- 1NF prohíbe relaciones anidadas

FN basadas en PK - 1FN

- **Ejemplo.**

EMP_PROY

Proyectos			
<u>E_CUIL</u>	<u>E_Nombre</u>	<u>P_Número</u>	<u>Horas</u>
27-23345876-9	Diego	1	20,5
		2	3,5
20-17283948-4	Laura	4	10
		2	7,5
27-38476827-2	Marina	4	11,5
		7	3,0
	

- E_CUIL es PK de EMP_PROY. P_Número es clave parcial de relación anidada

- **Técnica para alcanzar 1FN.**

- Mover atributos de relación anidada a una nueva relación
- Agregar a la nueva relación la PK de relación original
- PK de la nueva relación es: Clave parcial + PK relación original

EMP

EMP_PROY

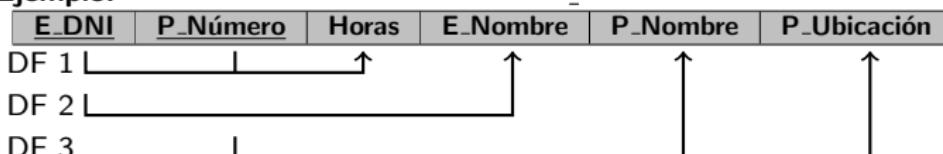
<u>E_CUIL</u>	<u>E_Nombre</u>
27-23345876-9	Diego
20-17283948-4	Laura
27-38476827-2	Marina

<u>E_CUIL</u>	<u>P_Número</u>	<u>Horas</u>
27-23345876-9	1	20,5
27-23345876-9	2	3,5
20-17283948-4	4	10
27-38476827-2	2	7,5
27-38476827-2	4	11,5
27-38476827-2	7	3,0
...

FN basadas en PK - 2FN

- **DF Completa.** Una DF $X \rightarrow Y$ es Completa si al eliminar algún atributo A de X la DF deja de existir
- **DF Parcial.** Una DF $X \rightarrow Y$ es Parcial si es posible eliminar algún atributo A de X y la DF continúa existiendo
- **Ejemplo.**

EMPLEADO _ PROYECTO



- Horas depende de manera Completa de PK
- E.Nombre depende de manera Parcial de PK
- P.Nombre y P_Ubicación dependen de manera Parcial de PK
- **2FN.** Un esquema R está en 2FN si todo atributo no primo A de R depende funcionalmente de manera completa de la PK de R
- **Tips.**

- Verificar sólo DFs cuyos lado izq. posean atributos que sean parte de la PK; si la PK es un solo atributo no es necesario realizar ningún test. ¿Por qué?

FN basadas en PK - 2FN

- Ejemplo.

EMPLEADO_PROYECTO					
<u>E_DNI</u>	<u>P_Número</u>	Horas	<u>E.Nombre</u>	<u>P.Nombre</u>	<u>P.Ubicación</u>
DF 1			↑		
DF 2				↑	
DF 3		↑			↑

- ¿Está en 2FN? ¡NO! Se ve, por DF 2 y DF 3, que hay atributos que dependen parcialmente de la PK

- Decomposición en 2FN

EP1

<u>E_DNI</u>	<u>P_Número</u>	Horas
DF 1		↑

EP2

<u>E_DNI</u>	<u>E.Nombre</u>
DF 2	↑

EP3

<u>P.Número</u>	<u>P.Nombre</u>	<u>P.Ubicación</u>
DF 3	↑	↑

FN basadas en PK - 3FN

- **Dependencia Transitiva** Una DF $X \rightarrow Y$ en R es Transitiva, si existe un conjunto de atributos Z en R que no son ni Clave Candidata ni un subconjunto de alguna Clave de R , tal que $X \rightarrow Z$ y $Z \rightarrow Y$
- **Ejemplo.**



La DF $E_CUIL \rightarrow D_Nombre$ es transitiva a través de Nro_Depto ya que:

- Existe $E_CUIL \rightarrow Nro_Depto$
- Existe $Nro_Depto \rightarrow D_Nombre$
- Nro_Depto no es ni clave candidata ni parte de una clave de **EMPLEADO_DEPARTAMENTO**
- **3FN.** Un esquema R está en 3FN si está en 2FN y ningún atributo no primo de R depende transitivamente de la PK

FN basadas en PK - 3FN

- Ejemplo.

EMPLEADO_DEPARTAMENTO

E_Nombre	E_CUIL	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
DFs	↑	↑	↑	↑

- ¿Está en 2FN? ¡Sí! No hay dependencias parciales sobre la PK
- ¿Está en 3FN? ¡NO! \exists dependencia transitiva $E_CUIL \rightarrow D_Nombre$

- Descomposición en 3FN.

ED1				
E_Nombre	E_CUIL	E_Fecha_Nacimiento	Nro_Depto	
DFs	↑	↑	↑	

ED2	
Nro_Depto	D_Nombre
DFs	↑

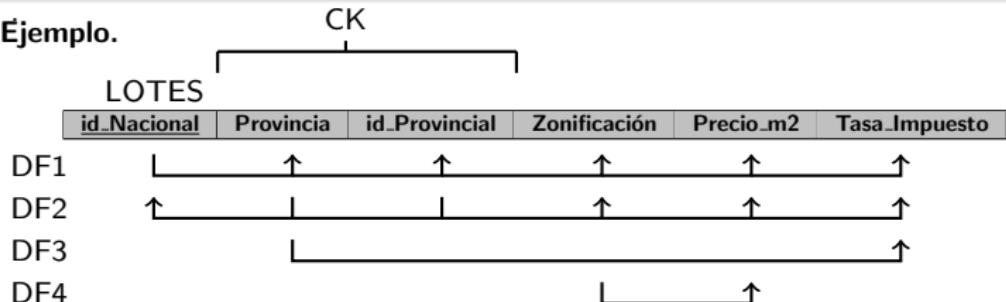
- $ED1 \bowtie ED2$ recomponen *EMPLEADO_DEPARTAMENTO* sin generar tuplas espúreas

Definición General - 2FN

- **2FN.** Un esquema R está en 2FN si todo atributo no primo A de R no depende parcialmente (de manera funcional) de ninguna clave de R
- **2FN. Definición Alternativa.** Un esquema R está en 2FN si todo atributo no primo A de R depende completamente (de manera funcional) de todas las claves de R

Definición General - 2FN

- Ejemplo.



- ¿Está en 2FN? ¡No! Tasa_Impuesto depende parcialmente de una CK (ver DF3)
- Descomposición en 2FN.

LOTES_1

	<u>id_Nacional</u>	Provincia	id_Provincial	Zonificación	Precio_m2
DF1	l	↑	↑	↑	↑
DF2	↑	l	l	↑	↑
DF4				l	↑

LOTES_2

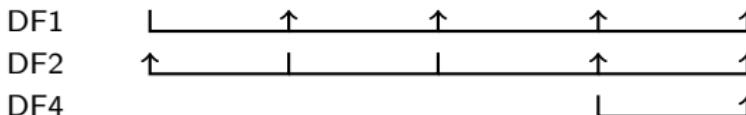
	Provincia	Tasa_Impuesto
DF3	l	↑

Definición General - 3FN

- **3FN.** Un esquema R está en 3FN si, para toda dependencia funcional *no trivial* $X \rightarrow A$ de R , se cumple alguna de las siguientes condiciones:
 - X es SK de R
 - A es atributo primo de R
- **DF trivial.** La DF $A \rightarrow B$ es trivial si B es un subconjunto de atributos de A .
 Ej. $A \rightarrow A$ es una DF trivial
- **Ejemplo.**

LOTES_1

	<u>id_Nacional</u>	Provincia	<u>id_Provincial</u>	Zonificación	Precio_m2
--	--------------------	-----------	----------------------	--------------	-----------



LOTES_2

	<u>Provincia</u>	Tasa.Impuesto
--	------------------	---------------



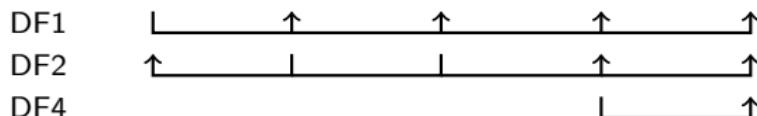
- ¿LOTES_1 está en 3FN? ¡No! Debido a DF 4
- ¿LOTES_2 está en 3FN? ¡Sí! Provincia es SK

Definición General - 3FN

● Ejemplo.

LOTES_1

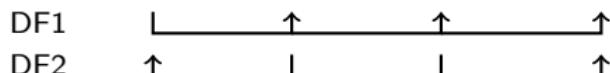
	<u>id_Nacional</u>	Provincia	<u>id_Provincial</u>	Zonificación	Precio_m2
--	--------------------	-----------	----------------------	--------------	-----------



● Descomposición en 3FN.

LOTES_1A

	<u>id_Nacional</u>	Provincia	<u>id_Provincial</u>	Zonificación
--	--------------------	-----------	----------------------	--------------



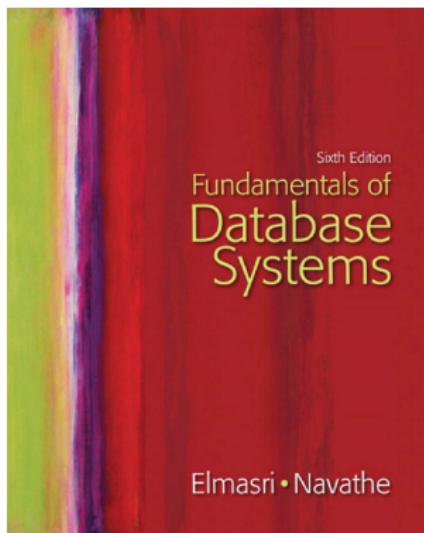
LOTES_1B

	Zonificación	Precio_m2
--	--------------	-----------



Normalización - Bibliografía

- Capítulo 15 (hasta 15.5 inclusive) Elmasri/Navathe - Fundamentals of Database Systems, 6th Ed., Pearson, 2011. (Capítulo 14 de la 7ma Ed.)



Normalización - Marco General

- **Salida del Diseño.** Conjunto de relaciones
- **Calidad de Diseño.** Necesidad de evaluar si una forma de agrupar atributos en un esquema es mejor que otra
- **Niveles.**
 - ① **Lógico (o Conceptual).** Un buen diseño de esquemas a este nivel habilita a los usuarios a entender el significado de los datos de las relaciones
 - ② **Implementación (o de Almacenamiento Físico).** Cómo se almacenan y actualizan las tuplas
- **Objetivos.**
 - **Preservar la Información.** Conceptos
 - **Minimizar Redundancia** Evitar almacenamiento de información redundante
- **Pautas de Diseño.** Cuatro pautas informales de diseño pueden utilizarse como medida para determinar la calidad de un diseño:
 - ① Estar seguro que semántica de atributos en esquemas es clara
 - ② Reducir la información redundante en tuplas
 - ③ Reducir la cantidad de valores NULL en tuplas
 - ④ Desactivar la posibilidad de generar tuplas espúreas
- **Independencia.** Estas pautas NO son siempre independientes unas de otras

Normalización - Pauta Nro. 1 - Semántica

- **Semántica.** Cuanto más fácil es explicar la semántica de los esquemas, mejor es el diseño.
- **Ejemplo.** **EMPLEADO_PROJECTO**

<u>E_Nombre</u>	<u>E_DNI</u>	<u>E_Fecha_Nacimiento</u>	Dirección	<u>P_Nombre</u>	<u>P_Número</u>
-----------------	--------------	---------------------------	-----------	-----------------	-----------------

¿Opinión?

- Mezcla atributos de EMPLEADO con PROYECTO.
- Desde el punto de vista de la lógica, puede ser correcto: puede usarse como vista.
- Deficiente en cuanto a la calidad respecto de la semántica de la relación (Pauta Nro. 1).
- **Ejemplo OK.** **EMPLEADO**

<u>E_Nombre</u>	<u>E_DNI</u>	<u>E_Fecha_Nacimiento</u>	Dirección_Laboral
PROYECTO			
<u>P_Nombre</u>	<u>P_Número</u>		
TRABAJA_EN			
<u>E_DNI</u>	<u>P_Número</u>		

- **Pauta Nro. 1**
 - Diseñar esquemas tal que sea fácil de explicar su significado
 - No combinar atributos de diversos tipos de entidades y relaciones en una misma relación

Normalización - Pauta Nro. 2 - Almacenamiento

- **Objetivo.** Minimizar espacio de almacenamiento a través del diseño
- **Ejemplo.** ¿Qué hacer para que este diseño ocupe menos espacio de almacenamiento?

Diseño “A”

EMPLEADO_DEPARTAMENTO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
...

Diseño “B”

EMPLEADO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto
Diego	20222333	11/12/1970	5
Laura	33456234	02/04/1985	5
Marina	45432345	23/07/2006	2
Santiago	24345345	18/02/1975	5
...

DEPARTAMENTO

Nro_Depto	D_Nombre
5	Publicidad y Promoción
2	Reclutamiento y Selección

- Diseño “A” almacena NATURAL JOIN de Diseño “B”.
- **Anomalías de Actualización.** Almacenar NATURAL JOINS introduce problemas adicionales. **Anomalías.** Inserción, Delección y Modificación.

Normalización - Pauta Nro. 2 - Almacenamiento

1. Anomalías de Inserción.

E.Nombre	E.DNI	E.Fecha_Nacimiento	Nro.Depto	D.Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
...

- ¿Qué sucede si se desea insertar un nuevo empleado y se desconoce ó aún no ha sido asignado a un Departamento?

Insertar nuevo empleado requiere incluir valores en atributos de departamento o NULL (si aún no ha sido asignado a ninguno)

E.Nombre	E.DNI	E.Fecha_Nacimiento	Nro.Depto	D.Nombre
...
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
Tamara	27354632	28/02/1979	NULL	NULL

- ¿Qué problema surge al insertar empleado asociado al Depto. 5?

Insertar nuevo empleado a departamento 5, requiere que los datos del departamento sean *consistentes* con el resto de los registros

E.Nombre	E.DNI	E.Fecha_Nacimiento	Nro.Depto	D.Nombre
...
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
Tamara	27354632	28/02/1979	5	Publicaciones y Prop.

Normalización - Pauta Nro. 2 - Almacenamiento

1. Anomalías de Inserción.

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
...

- ¿Es posible insertar un nuevo departamento que aún no posee empleados asignados? **¡No!**
 - ① NULL en campos de empleados viola la integridad de la entidad (NULL en atributo clave E_DNI)
 - ② Cuando se asigna el primer empleado a dicho depto. esta tupla ya no es mas necesaria.

Normalización - Pauta Nro. 2 - Almacenamiento

1. Anomalías de Delección.

E.Nombre	E.DNI	E.Fecha.Nacimiento	Nro.Depto	D.Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción

- ¿Qué consecuencia tiene eliminar el registro correspondiente a Marina?

E.Nombre	E.DNI	E.Fecha.Nacimiento	Nro.Depto	D.Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Santiago	24345345	18/02/1975	5	Publicidad y Promoción

Se pierde toda la información correspondiente al departamento 2

Normalización - Pauta Nro. 2 - Almacenamiento

1. Anomalías de Modificación.

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Diego	20222333	11/12/1970	5	Publicidad y Promoción
Laura	33456234	02/04/1985	5	Publicidad y Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad y Promoción

- ¿Qué sucede si se desea modificar “Publicidad y Promoción” por “Publicidad, Promoción y Comunicación Integral”
- Modificar el valor de un atributo de un departamento requiere modificar TODAS las tuplas de ese departamento. Caso contrario, se generan **inconsistencias**.

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Diego	20222333	11/12/1970	5	Publicidad, Promoción y Comunicación Integral
Laura	33456234	02/04/1985	5	Publicidad, Promoción
Marina	45432345	23/07/2006	2	Reclutamiento y Selección
Santiago	24345345	18/02/1975	5	Publicidad, Promoción

Normalización - Pauta Nro. 2 - Almacenamiento

● Pauta Nro. 2.

- Diseñar esquemas tal que no permitan anomalías de inserción, delección y modificación
- Si permiten anomalías, señalarlas claramente y asegurar que programas que actualizan BD operarán correctamente

● Performance.

- Notar que esta pauta puede ser violada en favor de la performance
- Ejemplo. Guardar en cada factura cuánto falta pagar (saldo). Esto claramente se puede recuperar “recorriendo” los pagos asociados a una factura, pero hay que hacerlo cada vez que un usuario pregunta cuánto debe un cliente determinado, y es una pregunta bastante frecuente. El costo de esto es que, cada vez que se paga una factura, o se anula un pago hay que ir a actualizar ese número
- En tal caso se debe señalar y actuar en consecuencia (Ej. triggers/store procedures que realicen automáticamente actualizaciones)

Normalización - Pauta Nro. 3 - NULLs

- **Esquemas.** Atributos no relacionados y agrupados en una misma tabla pueden generar múltiples NULLs en una misma tupla.
- **Ejemplo.**

EMPLEADO_DEPARTAMENTO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_Depto	D_Nombre
Santiago	24345345	18/02/1975	5	Publicidad y Promoción
Tamara	27354632	28/02/1979	NULL	NULL

- **Problemas.** ¿Qué sucede en cuanto espacio, semántica, JOIN?
 - Desperdicio espacio almacenamiento
 - JOINs (en presencia de NULLs, INNER JOIN produce distinto resultado vs. OUTER JOIN)
 - ¿Cómo se interpretan funciones de agregación (COUNT, SUM, etc.)?
 - Diversas interpretaciones de NULL
 - El resultado no aplica a la tupla. Ej. Nro_Registro_Conducir no aplica a menores
 - Valor conocido pero ausente. Ej. Fecha_Nacimiento de un empleado puede ser desconocida
 - Valor desconocido (No sabemos si existe). Ej. Un empleado puede que tenga telefono y no sabemos el valor o puede que no tenga.

Normalización - Pauta Nro. 3 - NULLs

● Pauta Nro. 3.

- Evitar asignar atributos a relaciones, cuando estos frecuentemente pueden ser NULLs
- Si NULLs son inevitables, asegurar que las situaciones son excepcionales y no aplican a la mayoría de las tuplas

Normalización - Pauta Nro. 4 - Tuplas Espúreas

- Ejemplo. Esquema original**

EMPLEADO_PROYECTO

E_Nombre	E_DNI	E_Fecha_Nacimiento	Nro_PROYECTO	P_Ubicación
Diego	20222333	11/12/1970	5	Argentina
Laura	33456234	02/04/1985	5	Argentina
Marina	45432345	23/07/2006	2	Uruguay
Santiago	24345345	18/02/1975	5	Argentina

- Descomposición.**

E_DNI	E_Fecha_Nacimiento	Nro_PROYECTO	P_Ubicación
20222333	11/12/1970	5	Argentina
33456234	02/04/1985	5	Argentina
45432345	23/07/2006	2	Uruguay
24345345	18/02/1975	5	Argentina

E_Nombre	P_Ubicación
Diego	Argentina
Laura	Argentina
Marina	Uruguay
Santiago	Argentina

- ¿Qué problema genera esta descomposición?

No permite recuperar información original de EMPLEADO_PROYECTO

- ¿Cuál es el resultado de aplicar NATURAL JOIN?

Produce **tuplas espúreas** (información no válida)

E_DNI	E_Fecha_Nacimiento	Nro_PROYECTO	P_Ubicación	E_Nombre
20222333	11/12/1970	5	Argentina	Diego
33456234	02/04/1985	5	Argentina	Diego
24345345	18/02/1975	5	Argentina	Diego

Normalización - Pauta Nro. 4 - Tuplas Espúreas

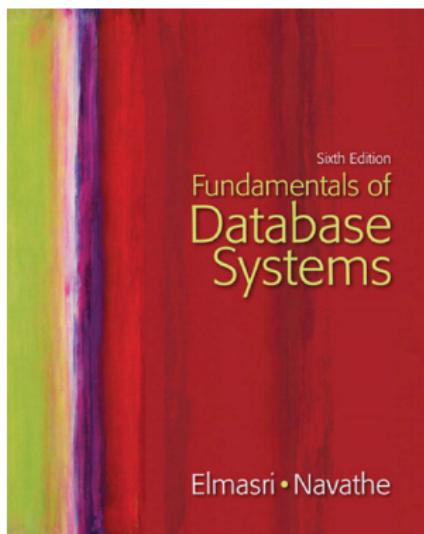
- **No deseable.** Esta descomposición no es deseable porque cuando se intenta la reconstrucción a través de NATURAL JOIN no se obtiene información correcta
- **Causa.** P_Ubicación, relaciona ambos esquemas, pero no es ni clave primaria ni clave foránea de ninguno de ellos
- **Pauta Nro. 4.**
 - Diseñar esquemas tal que puedan ser relacionados por atributos que se encuentren apropiadamente relacionados por medio de condiciones de igualdad entre ellos (clave primaria, clave foránea), para evitar generación de tuplas espúreas
 - Evitar relaciones que contengan atributos de matching que no sean combinación de claves foránea/clave primaria porque JOINS sobre ellos pueden producir tuplas espúreas

-Si tengo una tabla con 3 columnas y hago una partición para tener dos tablas:

-No puedo perder información (Sin Pérdida Información), pero puedo perder dependencia funcional.
-Tengo que partir por la key (si original tiene A,B,C y la key es A), parto en AB y AC.

Normalización - Bibliografía

- Capítulo 15 (hasta 15.5 inclusive) Elmasri/Navathe - Fundamentals of Database Systems, 6th Ed., Pearson, 2011. (Capítulo 14 de la 7ma Ed.)



Resumen Bases de Datos

Natalia Pierry

2023-09-12

Normalización: Pautas de Diseño

La normalización de bases de datos es un proceso importante en el diseño de bases de datos relacionales que consiste en designar y aplicar una serie de reglas a las relaciones obtenidas tras el paso del modelo entidad-relación al modelo relacional con el objetivo de:

- Minimizar la redundancia de los datos
- Disminuir problemas de actualización de los datos en las tablas
- Proteger la integridad de los datos.

Desde el nivel **lógico** (o conceptual, un buen diseño de esquemas habilita a los usuarios a entender el significado de los datos de las relaciones. Desde el punto de vista de la **implementación** (o de almacenamiento físico) habilita a ver cómo se almacenan y actualizan las tuplas.

Hay 4 pautas informales de diseño que pueden utilizarse como medida para determinar la calidad del mismo:

- Estar seguro de que la semántica de atributos es clara
- Reducir la información redundante en las tuplas
- Reducir la cantidad de valores null en las tuplas
- Desactivar la posibilidad de generar tuplas espúreas

(Estas pautas no siempre son independientes unas de otras)

Semántica Cuanto más fácil es explicar la semántica de los esquemas, mejor es el diseño. Se trata de diseñar esquemas tal que su significado sea fácil de explicar. No se deben combinar atributos de diversos tipos de entidades y relaciones en una misma relación.

Almacenamiento El objetivo es minimizar el espacio de almacenamiento a través del diseño. Si tenemos más de un valor en una columna y ese valor hace referencia a otras columnas, lo mejor es separarlo en dos tablas. Sin embargo, almacenar *natural joins* introduce problemas adicionales: anomalías de inserción, delección y modificación. Se intenta diseñar esquemas tal que no se permitan este tipo de anomalías y si se permiten, se deben señalar claramente y asegurar que los programas que actualizan las bases de datos operen correctamente en su presencia.

NULLs Atributos no relacionados y agrupados en una misma tabla pueden generar múltiples NULLs en una misma tupla. El gran problema es que el NULL tiene diversas interpretaciones: el resultado no aplica a la tupla, sabemos que existe el valor, pero está ausente, o podemos tener un valor desconocido que no sabemos si existe. Además de que se desperdicia espacio de almacenamiento, hay dos problemas fundamentales que son: los joins (inner/outer) producen resultados distintos y además no se sabe cómo se interpretan las funciones de agregación (count, sum, etc.). Entonces, se debe evitar asignar atributos a relaciones, cuando estos frecuentemente pueden ser NULLs. Si éstos son inevitables, se debe asegurar que las situaciones son excepcionales y no aplican a la mayoría de las tuplas.

Tuplas espúreas Se deben diseñar esquemas tal que puedan ser relacionados por atributos que se encuentren apropiadamente relacionados por medio de condiciones de igualdad entre ellos (clave primaria, clave foránea), para evitar la generación de tuplas espúreas. También se deben evitar relaciones que contengan atributos de matching que no sean combinación de clave foránea / clave primaria, porque joins sobre ellos pueden producir tuplas espúreas.

Normalización: Dependencias Funcionales

De la normalización (lógica) a la implementación (física o real) se sugiere tener dependencias funcionales para lograr la eficiencia en las tablas. La dependencia funcional es la base del proceso de normalización de bases de datos relacionales que garantiza la integridad de los datos. Por ejemplo, no vale la pena tener en la misma tabla el nombre y el DNI de una persona ya que refieren a lo mismo. A efectos prácticos, almacenar datos del estilo en una sola tabla resulta en redundancia y abre la puerta a inconsistencias de datos e inefficiencia en su uso.

El objetivo de la normalización de los datos es minimizar la redundancia y las anomalías de inserción, delección y modificación. Los esquemas que no pasan ciertos test de formas normales, se descomponen en esquemas más pequeños que sí pasan el test. La **forma normal** de una relación refiere a la mayor forma normal alcanzada por ella. Aún así, no garantizan un buen diseño de la base de datos si son consideradas aisladas de otros factores.

Luego del proceso de normalización por descomposición:

- Nonadditive Join (Lossless Join): es la garantía de que no ocurre el problema de generación de tuplas espúrias. La relación original tiene que poder ser recuperada de la descomposición. Lossless join debe lograrse *a cualquier costo*.
- Preservación: es la garantía de que cada dependencia funcional se encuentra representada en algún esquema resultante de la descomposición. Es deseable pero en algunos casos se puede sacrificar.

1FN - Relaciones Anidadas Prohibe relaciones dentro de relaciones, o relaciones como valores de atributos dentro de tuplas. El dominio de un atributo debe incluir sólo valores atómicos, es decir, la tupla puede tomar un solo valor del dominio. Hay que tener cuidado de que al intentar resolver un problema para que la tabla cumpla la 1FN no derive en redundancia o se generen valores NULLs. **Solución:** La mejor solución será remover el atributo que viola la 1FN de la tabla original y ubicarlo en una nueva relación junto con la PK de la relación original. Esta técnica se puede utilizar de forma recursiva para múltiples niveles, pero debe manejarse con cuidado.

- La dependencia funcional $X \rightarrow Y$ es *completa* si al eliminar un atributo de X la dependencia funcional deja de existir.
- La dependencia funcional $X \rightarrow Y$ es *parcial* si es posible eliminar un atributo de X y la dependencia funcional continúa existiendo.
- La dependencia funcional $X \rightarrow Y$ es *transitiva* si existe un conjunto de atributos Z que no son ni clave candidata ni un subconjunto de alguna clave tal que $X \rightarrow Z$ y $Z \rightarrow Y$

2FN - Dependencias Parciales Un esquema está en 2FN si todo atributo no primo depende *completamente* (de manera funcional) de la PK. Es decir, no debe haber atributos que dependan *parcialmente* (de manera funcional) de la PK.

3FN - Dependencia Transitiva Un esquema está en 3FN si está en 2FN y ningún atributo no primo depende *transitivamente* de la PK.

Desnormalizar

Una base de datos correctamente normalizada representa una buena estrategia de diseño, pero con frecuencia puede introducir mucha complejidad en cuanto a soporte, mantenimiento y nuevos desarrollos. Una base de datos bien diseñada puede significar que para obtener la información necesaria haya que atravesar muchas tablas que representan la información que se está buscando. Aunque hay muchas soluciones para este problema, como vistas, tablas temporales, y más, pero es importante no despreciar el valor de una desnormalización bien hecha. En este sentido, desnormalizar no es una estrategia de diseño, es una *solución de diseño*. La desnormalización es el proceso de procurar optimizar el funcionamiento de una base de datos por medio de agregar datos redundantes. Es decir, se duplica información en las tablas, violando las reglas de normalización.

Usualmente hay dos puntos en el ciclo de vida del desarrollo donde la desnormalización toma sentido:

1. **Durante el diseño inicial:** es muy fácil dejar de lado la simplicidad antes de plantear una solución. Algunas veces, durante la fase de diseño el sistema se vuelve diabólicamente complicado. Sin embargo, aún teniendo en cuenta sus defectos, la fase de diseño representa el punto más económico en el cual repensar y reescribir código. A medida que se avanza en el proceso de desarrollo es más y más caro hacer cambios al sistema.
2. **Posterior a la implementación:** una vez que se han encontrado los cuellos de botella del sistema, hay una nueva oportunidad de poner a punto el rendimiento. Esta fase representa una buena ocasión para recodificar y pensar si conviene mantener el sistema normalizado. En particular, es útil saber qué queries o stored procedures son accedidos con más frecuencia. Si el sistema cae dentro de la regla del 80-20 donde el 20% de los stored procedures son usados el 80% del tiempo, esto puede ser un punto de partida para considerar la desnormalización.

Cualquier oportunidad de evitar grandes tablas representa potencialmente una gran ganancia en performance, porque atravesar tablas enormes para obtener un dato particular, puede ser muy costoso y no necesariamente el optimizador podría elegir un buen plan. A partir de comprender cómo se utilizará la base de datos, podríamos pensar que hay una oportunidad para desnormalizar.

Es importante recordar que la desnormalización no es siempre lo más apropiado para hacer: solo porque tenga la oportunidad de desnormalizar no significa que deba hacerlo. Hay que tener las siguientes consideraciones en cuenta:

- *¿La desnormalización es redituible?* Si no ahorra tiempo de desarrollo, ejecución o simplificaciones de diseño, hay que evitarlas.
- *¿Vale la pena renunciar a la tabla normalizada?* Agregar columnas, índices y modificar el esquema, tiene sus costos. Algunos están relacionados a espacio en el disco y otros a recodificar la lógica del negocio.
- *Lo que la desnormalización ahorra en la simplificación de una consulta, podría ser costoso en el mantenimiento futuro.* Las relaciones que se introduzcan en el modelo podrían no ser evidentes para aquellos que desarrollan y trabajan con la base de datos.

Como se menciona al comienzo, desnormalizar no es una estrategia de diseño, es un rodeo, una forma de eludir un problema. Por lo tanto, una desnormalización debería usarse solo en casos de necesidad. Por ejemplo, para evitar problemas de mantenimiento. En tal caso, incluso con los problemas de mantenimiento que una desnormalización conlleva, quizás tenga sentido implementarla.

Si se elige desnormalizar, aquí van algunas reglas.

- *No desnormalizar deliberadamente.* No hacerlo como un capricho y sin un ajustado análisis de un costo-beneficio.
- *Documentar.* El mantenimiento de un sistema desnormalizado en producción, requiere un gran trabajo de documentación porque las desnormalizaciones frecuentemente no son intuitivas.
- *Encapsular las modificaciones de desnormalización en transacciones.* Poner las modificaciones en la lógica del negocio en el mismo lugar donde haya modificaciones previas y encapsularlas en la misma transacción. Así, si hay una falla, no se tendrá el problema de perder seguridad en la viabilidad de la desnormalización.

Document Databases

Es una base no-relacional que almacena los datos como documentos estructurados. Un documento es una colección de pares: nombre de campo y valor. Los valores pueden ser un valor simple o una estructura compleja como listas, otro documento o listas de documentos hijos.

La decisión más importante es si *incrustar o referenciar*, y será lo que determinará en grado de desnormalización de los documentos.

Referenciar En un documento se hace referencia a un ID o una lista de ID de otro documento.

Incrustar En un documento se incluyen todos los datos (en principio) de otro documento.

Wide Column: Casandra

Optimización

Una base de datos se almacena como una colección de archivos. Cada archivo contiene registros del mismo tipo y se divide en bloques de igual tamaño. La organización de archivos se refiere a la forma en que los datos son almacenados dentro de un archivo y las formas en las que puede accederse.

Hay dos tipos de archivos clásicos:

HeapFile Son registros sin orden. Al insertarse un registro, se lo agrega al final del archivo o en alguno de los bloques con espacio libre. Entonces, las operaciones requieren búsqueda lineal por todos los bloques del archivo. La ventaja es que la inserción es fácil. La desventaja es que la búsqueda y modificación es poco óptima.

SortedFile Son registros ordenados a partir de una clave de búsqueda. Al insertarse un registro se lo agrega ordenadamente, lo que puede provocar una reorganización en los bloques del archivo. En este tipo de archivos, mejoran las búsquedas desde la columna ordenada, pero el resto de las operaciones suele requerir una búsqueda lineal.

Índices Son estructuras adicionales que aceleran ciertas operaciones de búsqueda sobre tablas. Tienen mayor costo en operaciones de escritura, actualización y borrado, y mayor costo en espacio ocupado. Hay dos tipos clásicos de índices:

- B+: son árboles balanceados. puede ser clustered vs unclustered. Un BTree es un arbol ordenado.
- Hash:una tabla de hash almacena las claves de búsqueda. Cada posición de una tabla hash se asocia con un conjunto de registros. Por esta razón, cada posición suele llamarse “bucket” y los valores de hash, “índices bucket”.Un hash es un mapeo de un conjunto a otro. La única ventaja de Hash es la velocidad de acceso, pero depende de las colisiones.

Optimizador de Consultas

Dada una consulta, es deseable encontrar un plan de ejecución eficiente. Cuando un usuario formula una consulta, se analiza y envía esta consulta a un optimizador de consultas que utiliza información sobre el modo que se guardan los datos para producir un plan de ejecución eficiente para la evaluación de esa consulta. Un plan de ejecución es un detalle de las acciones que debe realizar el motor para la evaluación de la consulta, representado habitualmente como un árbol de operadores con anotaciones que contiene información detallada adicional sobre los métodos de acceso que se deben emplear, etc.

Analizar todo el espacio de búsqueda es costoso. Entonces, hay que evitar analizarlo por completo, pero es conveniente tener un plan eficiente:

1. Realiza un arbol canónico
2. Realiza modificaciones sobre el árbol. Buscan mejorar la performance de la consulta independientemente de la organización física. Involucran propiedades que permiten construir una consulta equivalente a la original.

Heurística: mediante *técnicas algebraicas* realizan árboles equivalentes y por lo teneral, mejoran la performance de las consultas

También hay *técnicas físicas*, que implican seleccionar implementaciones para los operadores basándose en cómo están organizados los archivos y las estricturas adicionales que existen. Utilizan el *catálogo*, que es información estadística de los datos. Se actualiza periódicamente y no siempre está sincronizado con los datos reales. Además, permite estimar la selectividad de los diferentes operadores.

Modelo de Costos

Para comparar planes se define un modelo de costos. El costo será expresado en cantidad de accesos a disco (lecturas + escrituras) y predomina sobre el tiempo de CPU. El costo de un plan será una estimación y se elegirá un plan con menor costo estimado.

Materialización Los resultados intermedios se guardan directamente en disco. El siguiente paso de la consulta deberá levantarlos de disco nuevamente.

Pipeline Las tuplas se van pasando al nodo superior del parbol mientras se continúa ejecutando la operación.

Natural Joins

Block Nested Loops join Por cada grupo de bloques R , recorro todo S . Además a R lo termino leyendo una sola vez en memoria.

Index Nested Loops Join Por cada tupla de R hago una búsqueda en el índice I . Además a R lo termino recorriendo una sola vez en memoria.

Sort Merge Join Se ordenan R y S (si alguno ya está ordenado, este costo no se considera). Se hace el *merge* entre R y S ordenados.

Hash Join Las tuplas de S_1 se comparan solamente con las de R_1 . Se necesitan, para M particiones, $M + 1$ bloques en memoria.

Control de Concurrencia

Cuando trabajemos con una base de datos en la cual muchos usuarios quieren utilizar datos, vamos a trabajar con transacciones. Las transacciones pueden escribir, leer o consolidar los datos (significa que la transacción queda persistida en la base de datos).

- $T_i \rightarrow \text{Transacción}$
- $W_i(x) \rightarrow T_i \text{escribex}$
- $R_i(x) \rightarrow T_i \text{leex}$
- $C_i \rightarrow \text{Commit(Consolidadosdatos)}$
- $J_i \rightarrow \text{Abort(rollback)}$
- $S \rightarrow \text{Start(timestampdatos)}$

La semántica del *Commit* implica que una vez que ocurrió un commit, todo lo que pasó antes de ese commit, de la transacción, queda persistido sin duda en la base de datos, no importa el mecanismo que se use, los datos quedan consolidados.

Otra cosa que puede pasar es un *Abort* de la transacción (roll back). El abort lo que hace es anular todo lo que se hizo antes de esa transacción. No se contempla nada de lo que se hizo bajo esa transacción.

Otra cosa que es importante es el *Start* que es cuando empieza la transacción y la forma de medirlo es con un time stamp.

Se representan distintas transacciones: roja, amarilla, marrón. El estado ideal es cuando las transacciones ocurren todas juntas, es decir, toda la transacción roja ocurre junta, luego la transacción amarilla y luego la marrón. A esto se lo denomina historia serial, porque viene de una serie.

En la realidad no pasa eso: uno de los módulos se llama *scheduler*: va recibiendo las transacciones que vienen de afuera y las va acomodando para tratar de cumplir con todo lo que le están pidiendo. Recibe todas las operaciones mezcladas y va viendo en cada momento del tiempo lo que viene, una por una y tiene que resolver qué hacer con las operaciones que vienen en función de lo que va recibiendo en cada instante.

Para el análisis, vamos a suponer que todas las transacciones ocurren sobre un solo i , pero en realidad al scheduler le llegan transacciones de distintos items y tiene que leer y escribir cosas distintas en cada uno. Pero cuando quiero leer y escribir el mismo, entra en conflicto. Entonces, vamos a poder decir que una historia es *legal* cuando el scheduler puede manejar las órdenes que recibe y va a devolver datos confiables y consistentes. Si esta historia deja los mismos valores que el orden serial, implica que esa transacción está bien: la transacción en orden serial es la medida para poner a prueba las historias. Si la historia permite separar los pasos de forma tal que sea igual que poner las operaciones en serie, se dice que esa historia es *serializada*: se puede correr en serie y hace lo mismo que la historia que no está intercalada. Este es el mundo real de las operaciones.

Supongamos que tenemos la siguiente historia:

$$H_1 = W_1(x), R_2(x), W_1(y), R_3(y), C_2, C_3, A_1$$

Esta es una historia no legal: lo que genera el abort al final es que todo lo que se hizo, hay que deshacerlo. Pero la T_2 ya leyó lo que había escrito la T_1 , y la T_3 ya leyó lo que había escrito la T_1 , y ambas commitearon, consolidando los datos. Esto que pasó acá se llama aborto en cascada porque si aborta la T_1 hace abortar a todas las otras y no solo eso sino que obliga a cambiar la semántica del commit que es que el dato ya está consolidado. Hay que evitar que pasen estas cosas. Para esto hay distintas políticas:

- Política pesimista: lo que hace es poner un montón de reglas para que estas cosas no pasen. Si una historia que cumple con todo el filtro de la política pesimista implica que no va a haber problemas.
- Política optimista: deja que pase la historia hasta que haya un problema y ahí es cuando actúa. Cuando llega el commit tiene que tomar una decisión.

Política Optimista

Read too late La siguiente historia representa un problema ya que si la someto a la prueba para ver si es serializada no la pasa porque tendría que haber leído antes de escribir.

$$H_1 = S_1(x), S_2(x), W_2(x), R_1(x)$$

Write too late Aplicando la regla de ver si esta historia es legal, serializada (debería poder ejecutarse en serie). La segunda transacción lee y después la que empezó antes, la escribe y, en realidad debería escribir antes.

$$H_2 = S_1(x), S_2(x), R_2(x), W_1(x)$$

Grafos

- Es una base de datos, no es una visualización.
- Es una base de datos *transaccional*, es decir, cumple con las características del paradigma ACID.
- Guarda los datos en forma de *grafos*, network (no tablas).
- Las *relaciones* se representan mediante las aristas o líneas que unen los vértices. Son dirigidas y pueden, eventualmente, tener propiedades asociadas. El tipo de relación provee de un predicado mientras que la dirección de una relación muestra el sujeto y el objeto.
- Los *nodos* son los vértices que unen las relaciones. Típicamente representa una entidad. Opcionalmente se le pueden agregar etiquetas que indiquen el rol del nodo dentro del grafo.
- Un *camino* está definido por un conjunto de vértices interconectados por aristas.
- Las *propiedades* son características adicionales que se le pueden asignar tanto a los nodos como a las relaciones para otorgar información adicional y enriquecer el modelo.
- El modelo utilizado es el *etiquetado*, que se compone de los nodos y sus relaciones.
- Los grafos llevan incorporadas *etiquetas*, que pueden definir los distintos vértices y también las relaciones entre ellos. Con las etiquetas se pueden asignar propiedades tanto a los nodos como a las relaciones.

- El *lenguaje declarativo* nos dice qué es lo que queremos, no cómo accedemos a los datos.
- Cada nodo puede tener propiedades.
- Los nodos pueden ser etiquetados con una o más etiquetas.
- Las relaciones también pueden tener nombre y propiedades.

Chapter 1

The Worlds of Database Systems

Databases today are essential to every business. They are used to maintain internal records, to present data to customers and clients on the World-Wide-Web, and to support many other commercial processes. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators of the human genome, and by bio-chemists exploring the medicinal properties of proteins, along with many other scientists.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a *database management system*, or *DBMS*, or more colloquially a “database system.” A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely. These systems are among the most complex types of software available. The capabilities that a DBMS provides the user are:

1. *Persistent storage.* Like a file system, a DBMS supports the storage of very large amounts of data that exists independently of any processes that are using the data. However, the DBMS goes far beyond the file system in providing flexibility, such as data structures that support efficient access to very large amounts of data.
2. *Programming interface.* A DBMS allows the user or an application program to access and modify data through a powerful query language. Again, the advantage of a DBMS over a file system is the flexibility to manipulate stored data in much more complex ways than the reading and writing of files.
3. *Transaction management.* A DBMS supports concurrent access to data, i.e., simultaneous access by many distinct processes (called “transac-

tions") at once. To avoid some of the undesirable consequences of simultaneous access, the DBMS supports *isolation*, the appearance that transactions execute one-at-a-time, and *atomicity*, the requirement that transactions execute either completely or not at all. A DBMS also supports *durability*, the ability to recover from failures or errors of many types.

1.1 The Evolution of Database Systems

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schema* (logical structure of the data), using a specialized language called a *data-definition language*.
2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
3. Support the storage of very large amounts of data — many gigabytes or more — over a long period of time, keeping it secure from accident or unauthorized use and allowing efficient access to the data for queries and database modifications.
4. Control access to data from many users at once, without allowing the actions of one user to affect other users and without allowing simultaneous accesses to corrupt the data accidentally.

1.1.1 Early Database Management Systems

The first commercial database management systems appeared in the late 1960’s. These systems evolved from file systems, which provide some of item (3) above; file systems store data over a long period of time, and they allow the storage of large amounts of data. However, file systems do not generally guarantee that data cannot be lost if it is not backed up, and they don’t support efficient access to data items whose location in a particular file is not known.

Further, file systems do not directly support item (2), a query language for the data in files. Their support for (1) — a schema for the data — is limited to the creation of directory structures for files. Finally, file systems do not satisfy (4). When they allow concurrent access to files by several users or processes, a file system generally will not prevent situations such as two users modifying the same file at about the same time, so the changes made by one user fail to appear in the file.

The first important applications of DBMS's were ones where data was composed of many small items, and many queries or modifications were made. Here are some of these applications.

Airline Reservations Systems

In this type of system, the items of data include:

1. Reservations by a single customer on a single flight, including such information as assigned seat or meal preference.
2. Information about flights — the airports they fly from and to, their departure and arrival times, or the aircraft flown, for example.
3. Information about ticket prices, requirements, and availability.

Typical queries ask for flights leaving around a certain time from one given city to another, what seats are available, and at what prices. Typical data modifications include the booking of a flight for a customer, assigning a seat, or indicating a meal preference. Many agents will be accessing parts of the data at any given time. The DBMS must allow such concurrent accesses, prevent problems such as two agents assigning the same seat simultaneously, and protect against loss of records if the system suddenly fails.

Banking Systems

Data items include names and addresses of customers, accounts, loans, and their balances, and the connection between customers and their accounts and loans, e.g., who has signature authority over which accounts. Queries for account balances are common, but far more common are modifications representing a single payment from, or deposit to, an account.

As with the airline reservation system, we expect that many tellers and customers (through ATM machines or the Web) will be querying and modifying the bank's data at once. It is vital that simultaneous accesses to an account not cause the effect of a transaction to be lost. Failures cannot be tolerated. For example, once the money has been ejected from an ATM machine, the bank must record the debit, even if the power immediately fails. On the other hand, it is not permissible for the bank to record the debit and then not deliver the money if the power fails. The proper way to handle this operation is far from obvious and can be regarded as one of the significant achievements in DBMS architecture.

Corporate Records

Many early applications concerned corporate records, such as a record of each sale, information about accounts payable and receivable, or information about employees — their names, addresses, salary options, tax status, and

so on. Queries include the printing of reports such as accounts receivable or employees' weekly paychecks. Each sale, purchase, bill, receipt, employee hired, fired, or promoted, and so on, results in a modification to the database.

The early DBMS's, evolving from file systems, encouraged the user to visualize data much as it was stored. These database systems used several different data models for describing the structure of the information in a database, chief among them the "hierarchical" or tree-based model and the graph-based "network" model. The latter was standardized in the late 1960's through a report of CODASYL (Committee on Data Systems and Languages).¹

A problem with these early models and systems was that they did not support high-level query languages. For example, the CODASYL query language had statements that allowed the user to jump from data element to data element, through a graph of pointers among these elements. There was considerable effort needed to write such programs, even for very simple queries.

1.1.2 Relational Database Systems

Following a famous paper written by Ted Codd in 1970,² database systems changed significantly. Codd proposed that database systems should present the user with a view of data organized as tables called *relations*. Behind the scenes, there might be a complex data structure that allowed rapid response to a variety of queries. But, unlike the user of earlier database systems, the user of a relational system would not be concerned with the storage structure. Queries could be expressed in a very high-level language, which greatly increased the efficiency of database programmers.

We shall cover the relational model of database systems throughout most of this book, starting with the basic relational concepts in Chapter 3. SQL ("Structured Query Language"), the most important query language based on the relational model, will be covered starting in Chapter 6. However, a brief introduction to relations will give the reader a hint of the simplicity of the model, and an SQL sample will suggest how the relational model promotes queries written at a very high level, avoiding details of "navigation" through the database.

Example 1.1: Relations are tables. Their columns are headed by *attributes*, which describe the entries in the column. For instance, a relation named **Accounts**, recording bank accounts, their balance, and type might look like:

<i>accountNo</i>	<i>balance</i>	<i>type</i>
12345	1000.00	savings
67890	2846.92	checking
...

¹ CODASYL Data Base Task Group April 1971 Report, ACM, New York.

²Codd, E. F., "A relational model for large shared data banks," *Comm. ACM*, **13**:6, pp. 377–387.

Heading the columns are the three attributes: `accountNo`, `balance`, and `type`. Below the attributes are the rows, or *tuples*. Here we show two tuples of the relation explicitly, and the dots below them suggest that there would be many more tuples, one for each account at the bank. The first tuple says that account number 12345 has a balance of one thousand dollars, and it is a savings account. The second tuple says that account 67890 is a checking account with \$2846.92.

Suppose we wanted to know the balance of account 67890. We could ask this query in SQL as follows:

```
SELECT balance
FROM Accounts
WHERE accountNo = 67890;
```

For another example, we could ask for the savings accounts with negative balances by:

```
SELECT accountNo
FROM Accounts
WHERE type = 'savings' AND balance < 0;
```

We do not expect that these two examples are enough to make the reader an expert SQL programmer, but they should convey the high-level nature of the SQL “select-from-where” statement. In principle, they ask the DBMS to

1. Examine all the tuples of the relation `Accounts` mentioned in the `FROM` clause,
2. Pick out those tuples that satisfy some criterion indicated in the `WHERE` clause, and
3. Produce as an answer certain attributes of those tuples, as indicated in the `SELECT` clause.

In practice, the system must “optimize” the query and find an efficient way to answer the query, even though the relations involved in the query may be very large. □

By 1990, relational database systems were the norm. Yet the database field continues to evolve, and new issues and approaches to the management of data surface regularly. In the balance of this section, we shall consider some of the modern trends in database systems.

1.1.3 Smaller and Smaller Systems

Originally, DBMS's were large, expensive software systems running on large computers. The size was necessary, because to store a gigabyte of data required a large computer system. Today, many gigabytes fit on a single disk, and

it is quite feasible to run a DBMS on a personal computer. Thus, database systems based on the relational model have become available for even very small machines, and they are beginning to appear as a common tool for computer applications, much as spreadsheets and word processors did before them.

1.1.4 Bigger and Bigger Systems

On the other hand, a gigabyte isn't much data. Corporate databases often occupy hundreds of gigabytes. Further, as storage becomes cheaper people find new reasons to store greater amounts of data. For example, retail chains often store *terabytes* (a terabyte is 1000 gigabytes, or 10^{12} bytes) of information recording the history of every sale made over a long period of time (for planning inventory; we shall have more to say about this matter in Section 1.1.7).

Further, databases no longer focus on storing simple data items such as integers or short character strings. They can store images, audio, video, and many other kinds of data that take comparatively huge amounts of space. For instance, an hour of video consumes about a gigabyte. Databases storing images from satellites can involve *petabytes* (1000 terabytes, or 10^{15} bytes) of data.

Handling such large databases required several technological advances. For example, databases of modest size are today stored on arrays of disks, which are called *secondary storage devices* (compared to main memory, which is "primary" storage). One could even argue that what distinguishes database systems from other software is, more than anything else, the fact that database systems routinely assume data is too big to fit in main memory and must be located primarily on disk at all times. The following two trends allow database systems to deal with larger amounts of data, faster.

Tertiary Storage

The largest databases today require more than disks. Several kinds of *tertiary storage devices* have been developed. Tertiary devices, perhaps storing a terabyte each, require much more time to access a given item than does a disk. While typical disks can access any item in 10-20 milliseconds, a tertiary device may take several seconds. Tertiary storage devices involve transporting an object, upon which the desired data item is stored, to a reading device. This movement is performed by a robotic conveyance of some sort.

For example, compact disks (CD's) or digital versatile disks (DVD's) may be the storage medium in a tertiary device. An arm mounted on a track goes to a particular disk, picks it up, carries it to a reader, and loads the disk into the reader.

Parallel Computing

The ability to store enormous volumes of data is important, but it would be of little use if we could not access large amounts of that data quickly. Thus, very large databases also require speed enhancers. One important speedup is

through index structures, which we shall mention in Section 1.2.2 and cover extensively in Chapter 13. Another way to process more data in a given time is to use parallelism. This parallelism manifests itself in various ways.

For example, since the rate at which data can be read from a given disk is fairly low, a few megabytes per second, we can speed processing if we use many disks and read them in parallel (even if the data originates on tertiary storage, it is “cached” on disks before being accessed by the DBMS). These disks may be part of an organized parallel machine, or they may be components of a distributed system, in which many machines, each responsible for a part of the database, communicate over a high-speed network when needed.

Of course, the ability to move data quickly, like the ability to store large amounts of data, does not by itself guarantee that queries can be answered quickly. We still need to use algorithms that break queries up in ways that allow parallel computers or networks of distributed computers to make effective use of all the resources. Thus, parallel and distributed management of very large databases remains an active area of research and development; we consider some of its important ideas in Section 15.9.

1.1.5 Client-Server and Multi-Tier Architectures

Many varieties of modern software use a *client-server* architecture, in which requests by one process (the *client*) are sent to another process (the *server*) for execution. Database systems are no exception, and it has become increasingly common to divide the work of a DBMS into a server process and one or more client processes.

In the simplest client/server architecture, the entire DBMS is a server, except for the query interfaces that interact with the user and send queries or other commands across to the server. For example, relational systems generally use the SQL language for representing requests from the client to the server. The database server then sends the answer, in the form of a table or relation, back to the client. The relationship between client and server can get more complex, especially when answers are extremely large. We shall have more to say about this matter in Section 1.1.6.

There is also a trend to put more work in the client, since the server will be a bottleneck if there are many simultaneous database users. In the recent proliferation of system architectures in which databases are used to provide dynamically-generated content for Web sites, the two-tier (client-server) architecture gives way to three (or even more) tiers. The DBMS continues to act as a server, but its client is typically an *application server*, which manages connections to the database, transactions, authorization, and other aspects. Application servers in turn have clients such as Web servers, which support end-users or other applications.

1.1.6 Multimedia Data

Another important trend in database systems is the inclusion of multimedia data. By “multimedia” we mean information that represents a signal of some sort. Common forms of multimedia data include video, audio, radar signals, satellite images, and documents or pictures in various encodings. These forms have in common that they are much larger than the earlier forms of data — integers, character strings of fixed length, and so on — and of vastly varying sizes.

The storage of multimedia data has forced DBMS’s to expand in several ways. For example, the operations that one performs on multimedia data are not the simple ones suitable for traditional data forms. Thus, while one might search a bank database for accounts that have a negative balance, comparing each balance with the real number 0.0, it is not feasible to search a database of pictures for those that show a face that “looks like” a particular image.

To allow users to create and use complex data operations such as image-processing, DBMS’s have had to incorporate the ability of users to introduce functions of their own choosing. Often, the object-oriented approach is used for such extensions, even in relational systems, which are then dubbed “object-relational.” We shall take up object-oriented database programming in various places, including Chapters 4 and 9.

The size of multimedia objects also forces the DBMS to modify the storage manager so that objects or tuples of a gigabyte or more can be accommodated. Among the many problems that such large elements present is the delivery of answers to queries. In a conventional, relational database, an answer is a set of tuples. These tuples would be delivered to the client by the database server as a whole.

However, suppose the answer to a query is a video clip a gigabyte long. It is not feasible for the server to deliver the gigabyte to the client as a whole. For one reason it takes too long and will prevent the server from handling other requests. For another, the client may want only a small part of the film clip, but doesn’t have a way to ask for exactly what it wants without seeing the initial portion of the clip. For a third reason, even if the client wants the whole clip, perhaps in order to play it on a screen, it is sufficient to deliver the clip at a fixed rate over the course of an hour (the amount of time it takes to play a gigabyte of compressed video). Thus, the storage system of a DBMS supporting multimedia data has to be prepared to deliver answers in an interactive mode, passing a piece of the answer to the client on request or at a fixed rate.

1.1.7 Information Integration

As information becomes ever more essential in our work and play, we find that existing information resources are being used in many new ways. For instance, consider a company that wants to provide on-line catalogs for all its products, so that people can use the World Wide Web to browse its products and place on-

line orders. A large company has many divisions. Each division may have built its own database of products independently of other divisions. These divisions may use different DBMS's, different structures for information, perhaps even different terms to mean the same thing or the same term to mean different things.

Example 1.2: Imagine a company with several divisions that manufacture disks. One division's catalog might represent rotation rate in revolutions per second, another in revolutions per minute. Another might have neglected to represent rotation speed at all. A division manufacturing floppy disks might refer to them as "disks," while a division manufacturing hard disks might call *them* "disks" as well. The number of tracks on a disk might be referred to as "tracks" in one division, but "cylinders" in another. □

Central control is not always the answer. Divisions may have invested large amounts of money in their database long before information integration across divisions was recognized as a problem. A division may have been an independent company, recently acquired. For these or other reasons, these so-called *legacy databases* cannot be replaced easily. Thus, the company must build some structure on top of the legacy databases to present to customers a unified view of products across the company.

One popular approach is the creation of *data warehouses*, where information from many legacy databases is copied, with the appropriate translation, to a central database. As the legacy databases change, the warehouse is updated, but not necessarily instantaneously updated. A common scheme is for the warehouse to be reconstructed each night, when the legacy databases are likely to be less busy.

The legacy databases are thus able to continue serving the purposes for which they were created. New functions, such as providing an on-line catalog service through the Web, are done at the data warehouse. We also see data warehouses serving needs for planning and analysis. For example, company analysts may run queries against the warehouse looking for sales trends, in order to better plan inventory and production. *Data mining*, the search for interesting and unusual patterns in data, has also been enabled by the construction of data warehouses, and there are claims of enhanced sales through exploitation of patterns discovered in this way. These and other issues of information integration are discussed in Chapter 20.

1.2 Overview of a Database Management System

In Fig. 1.1 we see an outline of a complete DBMS. Single boxes represent system components, while double boxes represent in-memory data structures. The solid lines indicate control and data flow, while dashed lines indicate data flow only.

Since the diagram is complicated, we shall consider the details in several stages. First, at the top, we suggest that there are two distinct sources of commands to the DBMS:

1. Conventional users and application programs that ask for data or modify data.
2. A *database administrator*: a person or persons responsible for the structure or *schema* of the database.

1.2.1 Data-Definition Language Commands

The second kind of command is the simpler to process, and we show its trail beginning at the upper right side of Fig. 1.1. For example, the database administrator, or *DBA*, for a university registrar’s database might decide that there should be a table or relation with columns for a student, a course the student has taken, and a grade for that student in that course. The DBA might also decide that the only allowable grades are A, B, C, D, and F. This structure and constraint information is all part of the schema of the database. It is shown in Fig. 1.1 as entered by the DBA, who needs special authority to execute schema-altering commands, since these can have profound effects on the database. These schema-altering *DDL commands* (“DDL” stands for “data-definition language”) are parsed by a DDL processor and passed to the execution engine, which then goes through the index/file/record manager to alter the *metadata*, that is, the schema information for the database.

1.2.2 Overview of Query Processing

The great majority of interactions with the DBMS follow the path on the left side of Fig. 1.1. A user or an application program initiates some action that does not affect the schema of the database, but may affect the content of the database (if the action is a modification command) or will extract data from the database (if the action is a query). Remember from Section 1.1 that the language in which these commands are expressed is called a data-manipulation language (*DML*) or somewhat colloquially a query language. There are many data-manipulation languages available, but SQL, which was mentioned in Example 1.1, is by far the most commonly used. DML statements are handled by two separate subsystems, as follows.

Answering the query

The query is parsed and optimized by a *query compiler*. The resulting *query plan*, or sequence of actions the DBMS will perform to answer the query, is passed to the *execution engine*. The execution engine issues a sequence of requests for small pieces of data, typically records or tuples of a relation, to a resource manager that knows about *data files* (holding relations), the format

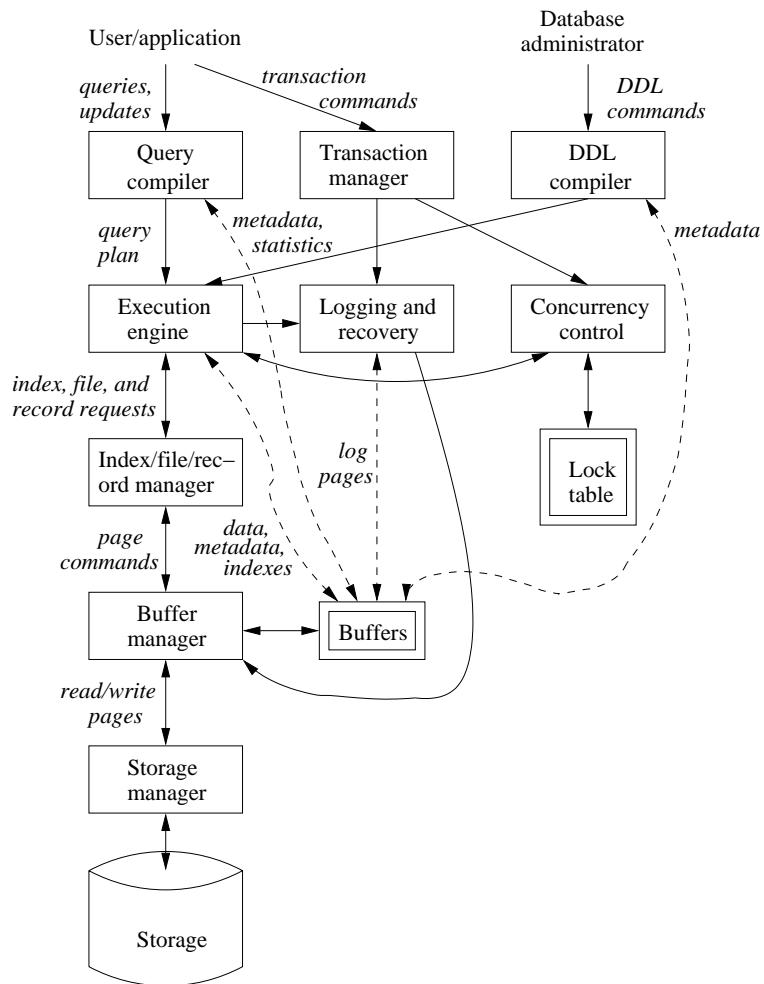


Figure 1.1: Database management system components

and size of records in those files, and *index files*, which help find elements of data files quickly.

The requests for data are translated into pages and these requests are passed to the *buffer manager*. We shall discuss the role of the buffer manager in Section 1.2.3, but briefly, its task is to bring appropriate portions of the data from secondary storage (disk, normally) where it is kept permanently, to main-memory buffers. Normally, the page or “disk block” is the unit of transfer between buffers and disk.

The buffer manager communicates with a storage manager to get data from disk. The storage manager might involve operating-system commands, but more typically, the DBMS issues commands directly to the disk controller.

Transaction processing

Queries and other DML actions are grouped into *transactions*, which are units that must be executed atomically and in isolation from one another. Often each query or modification action is a transaction by itself. In addition, the execution of transactions must be *durable*, meaning that the effect of any completed transaction must be preserved even if the system fails in some way right after completion of the transaction. We divide the transaction processor into two major parts:

1. A *concurrency-control manager*, or *scheduler*, responsible for assuring atomicity and isolation of transactions, and
2. A *logging and recovery manager*, responsible for the durability of transactions.

We shall consider these components further in Section 1.2.4.

1.2.3 Storage and Buffer Management

The data of a database normally resides in secondary storage; in today’s computer systems “secondary storage” generally means magnetic disk. However, to perform any useful operation on data, that data must be in main memory. It is the job of the *storage manager* to control the placement of data on disk and its movement between disk and main memory.

In a simple database system, the storage manager might be nothing more than the file system of the underlying operating system. However, for efficiency purposes, DBMS’s normally control storage on the disk directly, at least under some circumstances. The *storage manager* keeps track of the location of files on the disk and obtains the block or blocks containing a file on request from the buffer manager. Recall that disks are generally divided into *disk blocks*, which are regions of contiguous storage containing a large number of bytes, perhaps 2^{12} or 2^{14} (about 4000 to 16,000 bytes).

The *buffer manager* is responsible for partitioning the available main memory into *buffers*, which are page-sized regions into which disk blocks can be

transferred. Thus, all DBMS components that need information from the disk will interact with the buffers and the buffer manager, either directly or through the execution engine. The kinds of information that various components may need include:

1. *Data*: the contents of the database itself.
2. *Metadata*: the database schema that describes the structure of, and constraints on, the database.
3. *Statistics*: information gathered and stored by the DBMS about data properties such as the sizes of, and values in, various relations or other components of the database.
4. *Indexes*: data structures that support efficient access to the data.

A more complete discussion of the buffer manager and its role appears in Section 15.7.

1.2.4 Transaction Processing

It is normal to group one or more database operations into a *transaction*, which is a unit of work that must be executed atomically and in apparent isolation from other transactions. In addition, a DBMS offers the guarantee of durability: that the work of a completed transaction will never be lost. The *transaction manager* therefore accepts *transaction commands* from an application, which tell the transaction manager when transactions begin and end, as well as information about the expectations of the application (some may not wish to require atomicity, for example). The transaction processor performs the following tasks:

1. *Logging*: In order to assure durability, every change in the database is logged separately on disk. The *log manager* follows one of several policies designed to assure that no matter when a system failure or “crash” occurs, a *recovery manager* will be able to examine the log of changes and restore the database to some consistent state. The log manager initially writes the log in buffers and negotiates with the buffer manager to make sure that buffers are written to disk (where data can survive a crash) at appropriate times.
2. *Concurrency control*: Transactions must appear to execute in isolation. But in most systems, there will in truth be many transactions executing at once. Thus, the scheduler (concurrency-control manager) must assure that the individual actions of multiple transactions are executed in such an order that the net effect is the same as if the transactions had in fact executed in their entirety, one-at-a-time. A typical scheduler does its work by maintaining *locks* on certain pieces of the database. These locks prevent two transactions from accessing the same piece of data in

The ACID Properties of Transactions

Properly implemented transactions are commonly said to meet the “ACID test,” where:

- “A” stands for “atomicity,” the all-or-nothing execution of transactions.
- “I” stands for “isolation,” the fact that each transaction must appear to be executed as if no other transaction is executing at the same time.
- “D” stands for “durability,” the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

The remaining letter, “C,” stands for “consistency.” That is, all databases have consistency constraints, or expectations about relationships among data elements (e.g., account balances may not be negative). Transactions are expected to preserve the consistency of the database. We discuss the expression of consistency constraints in a database schema in Chapter 7, while Section 18.1 begins a discussion of how consistency is maintained by the DBMS.

ways that interact badly. Locks are generally stored in a main-memory *lock table*, as suggested by Fig. 1.1. The scheduler affects the execution of queries and other database operations by forbidding the execution engine from accessing locked parts of the database.

3. *Deadlock resolution*: As transactions compete for resources through the locks that the scheduler grants, they can get into a situation where none can proceed because each needs something another transaction has. The transaction manager has the responsibility to intervene and cancel (“rollback” or “abort”) one or more transactions to let the others proceed.

1.2.5 The Query Processor

The portion of the DBMS that most affects the performance that the user sees is the *query processor*. In Fig. 1.1 the query processor is represented by two components:

1. The *query compiler*, which translates the query into an internal form called a *query plan*. The latter is a sequence of operations to be performed on the data. Often the operations in a query plan are implementations of

“relational algebra” operations, which are discussed in Section 5.2. The query compiler consists of three major units:

- (a) A *query parser*, which builds a tree structure from the textual form of the query.
- (b) A *query preprocessor*, which performs semantic checks on the query (e.g., making sure all relations mentioned by the query actually exist), and performing some tree transformations to turn the parse tree into a tree of algebraic operators representing the initial query plan.
- (c) A *query optimizer*, which transforms the initial query plan into the best available sequence of operations on the actual data.

The query compiler uses metadata and statistics about the data to decide which sequence of operations is likely to be the fastest. For example, the existence of an *index*, which is a specialized data structure that facilitates access to data, given values for one or more components of that data, can make one plan much faster than another.

2. The *execution engine*, which has the responsibility for executing each of the steps in the chosen query plan. The execution engine interacts with most of the other components of the DBMS, either directly or through the buffers. It must get the data from the database into buffers in order to manipulate that data. It needs to interact with the scheduler to avoid accessing data that is locked, and with the log manager to make sure that all database changes are properly logged.

1.3 Outline of Database-System Studies

Ideas related to database systems can be divided into three broad categories:

1. *Design of databases*. How does one develop a useful database? What kinds of information go into the database? How is the information structured? What assumptions are made about types or values of data items? How do data items connect?
2. *Database programming*. How does one express queries and other operations on the database? How does one use other capabilities of a DBMS, such as transactions or constraints, in an application? How is database programming combined with conventional programming?
3. *Database system implementation*. How does one build a DBMS, including such matters as query processing, transaction processing and organizing storage for efficient access?

How Indexes Are Implemented

The reader may have learned in a course on data structures that a hash table is a very efficient way to build an index. Early DBMS's did use hash tables extensively. Today, the most common data structure is called a *B-tree*; the “B” stands for “balanced.” A B-tree is a generalization of a balanced binary search tree. However, while each node of a binary tree has up to two children, the B-tree nodes have a large number of children. Given that B-trees normally reside on disk rather than in main memory, the B-tree is designed so that each node occupies a full disk block. Since typical systems use disk blocks on the order of 2^{12} bytes (4096 bytes), there can be hundreds of pointers to children in a single block of a B-tree. Thus, search of a B-tree rarely involves more than a few levels.

The true cost of disk operations generally is proportional to the number of disk blocks accessed. Thus, searches of a B-tree, which typically examine only a few disk blocks, are much more efficient than would be a binary-tree search, which typically visits nodes found on many different disk blocks. This distinction, between B-trees and binary search trees, is but one of many examples where the most appropriate data structure for data stored on disk is different from the data structures used for algorithms that run in main memory.

1.3.1 Database Design

Chapter 2 begins with a high-level notation for expressing database designs, called the *entity-relationship model*. We introduce in Chapter 3 the relational model, which is the model used by the most widely adopted DBMS's, and which we touched upon briefly in Section 1.1.2. We show how to translate entity-relationship designs into relational designs, or “relational database schemas.” Later, in Section 6.6, we show how to render relational database schemas formally in the data-definition portion of the SQL language.

Chapter 3 also introduces the reader to the notion of “dependencies,” which are formally stated assumptions about relationships among tuples in a relation. Dependencies allow us to improve relational database designs, through a process known as “normalization” of relations.

In Chapter 4 we look at object-oriented approaches to database design. There, we cover the language ODL, which allows one to describe databases in a high-level, object-oriented fashion. We also look at ways in which object-oriented design has been combined with relational modeling, to yield the so-called “object-relational” model. Finally, Chapter 4 also introduces “semistructured data” as an especially flexible database model, and we see its modern embodiment in the document language XML.

1.3.2 Database Programming

Chapters 5 through 10 cover database programming. We start in Chapter 5 with an abstract treatment of queries in the relational model, introducing the family of operators on relations that form “relational algebra.”

Chapters 6 through 8 are devoted to SQL programming. As we mentioned, SQL is the dominant query language of the day. Chapter 6 introduces basic ideas regarding queries in SQL and the expression of database schemas in SQL. Chapter 7 covers aspects of SQL concerning constraints and triggers on the data.

Chapter 8 covers certain advanced aspects of SQL programming. First, while the simplest model of SQL programming is a stand-alone, generic query interface, in practice most SQL programming is embedded in a larger program that is written in a conventional language, such as C. In Chapter 8 we learn how to connect SQL statements with a surrounding program and to pass data from the database to the program’s variables and vice versa. This chapter also covers how one uses SQL features that specify transactions, connect clients to servers, and authorize access to databases by nonowners.

In Chapter 9 we turn our attention to standards for object-oriented database programming. Here, we consider two directions. The first, OQL (Object Query Language), can be seen as an attempt to make C++, or other object-oriented programming languages, compatible with the demands of high-level database programming. The second, which is the object-oriented features recently adopted in the SQL standard, can be viewed as an attempt to make relational databases and SQL compatible with object-oriented programming.

Finally, in Chapter 10, we return to the study of abstract query languages that we began in Chapter 5. Here, we study logic-based languages and see how they have been used to extend the capabilities of modern SQL.

1.3.3 Database System Implementation

The third part of the book concerns how one can implement a DBMS. The subject of database system implementation in turn can be divided roughly into three parts:

1. *Storage management*: how secondary storage is used effectively to hold data and allow it to be accessed quickly.
2. *Query processing*: how queries expressed in a very high-level language such as SQL can be executed efficiently.
3. *Transaction management*: how to support transactions with the ACID properties discussed in Section 1.2.4.

Each of these topics is covered by several chapters of the book.

Storage-Management Overview

Chapter 11 introduces the memory hierarchy. However, since secondary storage, especially disk, is so central to the way a DBMS manages data, we examine in the greatest detail the way data is stored and accessed on disk. The “block model” for disk-based data is introduced; it influences the way almost everything is done in a database system.

Chapter 12 relates the storage of data elements — relations, tuples, attribute-values, and their equivalents in other data models — to the requirements of the block model of data. Then we look at the important data structures that are used for the construction of indexes. Recall that an index is a data structure that supports efficient access to data. Chapter 13 covers the important one-dimensional index structures — indexed-sequential files, B-trees, and hash tables. These indexes are commonly used in a DBMS to support queries in which a value for an attribute is given and the tuples with that value are desired. B-trees also are used for access to a relation sorted by a given attribute. Chapter 14 discusses multidimensional indexes, which are data structures for specialized applications such as geographic databases, where queries typically ask for the contents of some region. These index structures can also support complex SQL queries that limit the values of two or more attributes, and some of these structures are beginning to appear in commercial DBMS’s.

Query-Processing Overview

Chapter 15 covers the basics of query execution. We learn a number of algorithms for efficient implementation of the operations of relational algebra. These algorithms are designed to be efficient when data is stored on disk and are in some cases rather different from analogous main-memory algorithms.

In Chapter 16 we consider the architecture of the query compiler and optimizer. We begin with the parsing of queries and their semantic checking. Next, we consider the conversion of queries from SQL to relational algebra and the selection of a *logical query plan*, that is, an algebraic expression that represents the particular operations to be performed on data and the necessary constraints regarding order of operations. Finally, we explore the selection of a *physical query plan*, in which the particular order of operations and the algorithm used to implement each operation have been specified.

Transaction-Processing Overview

In Chapter 17 we see how a DBMS supports durability of transactions. The central idea is that a log of all changes to the database is made. Anything that is in main-memory but not on disk can be lost in a crash (say, if the power supply is interrupted). Therefore we have to be careful to move from buffer to disk, in the proper order, both the database changes themselves and the log of what changes were made. There are several log strategies available, but each limits our freedom of action in some ways.

Then, we take up the matter of concurrency control — assuring atomicity and isolation — in Chapter 18. We view transactions as sequences of operations that read or write database elements. The major topic of the chapter is how to manage locks on database elements: the different types of locks that may be used, and the ways that transactions may be allowed to acquire locks and release their locks on elements. Also studied are a number of ways to assure atomicity and isolation without using locks.

Chapter 19 concludes our study of transaction processing. We consider the interaction between the requirements of logging, as discussed in Chapter 17, and the requirements of concurrency that were discussed in Chapter 18. Handling of deadlocks, another important function of the transaction manager, is covered here as well. The extension of concurrency control to a distributed environment is also considered in Chapter 19. Finally, we introduce the possibility that transactions are “long,” taking hours or days rather than milliseconds. A long transaction cannot lock data without causing chaos among other potential users of that data, which forces us to rethink concurrency control for applications that involve long transactions.

1.3.4 Information Integration Overview

Much of the recent evolution of database systems has been toward capabilities that allow different *data sources*, which may be databases and/or information resources that are not managed by a DBMS, to work together in a larger whole. We introduced you to these issues briefly, in Section 1.1.7. Thus, in the final Chapter 20, we study important aspects of information integration. We discuss the principal modes of integration, including translated and integrated copies of sources called a “data warehouse,” and virtual “views” of a collection of sources, through what is called a “mediator.”

1.4 Summary of Chapter 1

- ◆ *Database Management Systems:* A DBMS is characterized by the ability to support efficient access to large amounts of data, which persists over time. It is also characterized by support for powerful query languages and for durable transactions that can execute concurrently in a manner that appears atomic and independent of other transactions.
- ◆ *Comparison With File Systems:* Conventional file systems are inadequate as database systems, because they fail to support efficient search, efficient modifications to small pieces of data, complex queries, controlled buffering of useful data in main memory, or atomic and independent execution of transactions.
- ◆ *Relational Database Systems:* Today, most database systems are based on the relational model of data, which organizes information into tables. SQL is the language most often used in these systems.

- ◆ *Secondary and Tertiary Storage:* Large databases are stored on secondary storage devices, usually disks. The largest databases require tertiary storage devices, which are several orders of magnitude more capacious than disks, but also several orders of magnitude slower.
- ◆ *Client-Server Systems:* Database management systems usually support a client-server architecture, with major database components at the server and the client used to interface with the user.
- ◆ *Future Systems:* Major trends in database systems include support for very large “multimedia” objects such as videos or images and the integration of information from many separate information sources into a single database.
- ◆ *Database Languages:* There are languages or language components for defining the structure of data (data-definition languages) and for querying and modification of the data (data-manipulation languages).
- ◆ *Components of a DBMS:* The major components of a database management system are the storage manager, the query processor, and the transaction manager.
 - ◆ *The Storage Manager:* This component is responsible for storing data, metadata (information about the schema or structure of the data), indexes (data structures to speed the access to data), and logs (records of changes to the database). This material is kept on disk. An important storage-management component is the buffer manager, which keeps portions of the disk contents in main memory.
 - ◆ *The Query Processor:* This component parses queries, optimizes them by selecting a query plan, and executes the plan on the stored data.
 - ◆ *The Transaction Manager:* This component is responsible for logging database changes to support recovery after a system crashes. It also supports concurrent execution of transactions in a way that assures atomicity (a transaction is performed either completely or not at all), and isolation (transactions are executed as if there were no other concurrently executing transactions).

1.5 References for Chapter 1

Today, on-line searchable bibliographies cover essentially all recent papers concerning database systems. Thus, in this book, we shall not try to be exhaustive in our citations, but rather shall mention only the papers of historical importance and major secondary sources or useful surveys. One searchable index

of database research papers has been constructed by Michael Ley [5]. Alf-Christian Achilles maintains a searchable directory of many indexes relevant to the database field [1].

While many prototype implementations of database systems contributed to the technology of the field, two of the most widely known are the System R project at IBM Almaden Research Center [3] and the INGRES project at Berkeley [7]. Each was an early relational system and helped establish this type of system as the dominant database technology. Many of the research papers that shaped the database field are found in [6].

The 1998 “Asilomar report” [4] is the most recent in a series of reports on database-system research and directions. It also has references to earlier reports of this type.

You can find more about the theory of database systems than is covered here from [2], [8], and [9].

1. <http://liinwww.ira.uka.de/bibliography/Database> .
2. Abiteboul, S., R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.
3. M. M. Astrahan et al., “System R: a relational approach to database management,” *ACM Trans. on Database Systems* 1:2 (1976), pp. 97–137.
4. P. A. Bernstein et al., “The Asilomar report on database research,” http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/Asilomar_Final.htm .
5. <http://www.informatik.uni-trier.de/~ley/db/index.html> . A mirror site is found at <http://www.acm.org/sigmod/dblp/db/index.html> .
6. Stonebraker, M. and J. M. Hellerstein (eds.), *Readings in Database Systems*, Morgan-Kaufmann, San Francisco, 1998.
7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, “The design and implementation of INGRES,” *ACM Trans. on Database Systems* 1:3 (1976), pp. 189–222.
8. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.
9. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, New York, 1989.

Capítulo 1 de "The world of database systems" de Ullman (dia1)

Una base de datos es una colección de información que existe por un largo período de tiempo.

Las propiedades ACID son un conjunto de características fundamentales que garantizan la integridad y consistencia de los datos en una base de datos transaccional. Estas propiedades son esenciales en sistemas donde la precisión y la confiabilidad de la información son críticas. A continuación, cada una de las propiedades ACID:

- Atomicidad (Atomicity):

La propiedad de atomicidad garantiza que una transacción se realice como una unidad indivisible o "atómica". Esto significa que una transacción se ejecuta por completo o no se ejecuta en absoluto.

Si una transacción consiste en varias operaciones individuales (por ejemplo, inserciones, actualizaciones o eliminaciones de registros), todas estas operaciones se realizan con éxito o se anulan por completo en caso de error. No puede haber un estado intermedio en el que algunas operaciones se hayan completado y otras no.

- Consistencia (Consistency):

La propiedad de consistencia garantiza que una transacción lleve la base de datos de un estado válido a otro estado válido. Esto significa que una transacción no puede violar las reglas de integridad definidas en la base de datos.

Si una transacción se ejecuta de manera exitosa, la base de datos debe quedar en un estado coherente y válido en términos de sus restricciones y reglas de negocio.

- Aislamiento (Isolation):

La propiedad de aislamiento se refiere a que las transacciones en ejecución son invisibles entre sí. Esto impide que una transacción vea los cambios realizados por otras transacciones hasta que estas se completen.

El aislamiento evita problemas de concurrencia, como lecturas sucias, escrituras intercaladas y conflictos de lectura/escritura, que podrían ocurrir si varias transacciones acceden y modifican datos al mismo tiempo.

- Durabilidad (Durability):

La propiedad de durabilidad garantiza que una vez que una transacción se haya completado con éxito (habiendo cumplido con las propiedades anteriores), sus cambios persistirán en la base de datos incluso en caso de fallo del sistema, como una caída de energía o un reinicio del servidor.

La durabilidad asegura que los datos se almacenan de manera segura y no se pierden.

Artículo sobre el teorema CAP (Berkeley) de Eric Brewer (dia1)

[en ppt NoSQL_2022(dia1) hay info sobre esto también]

El teorema CAP establece que cualquier sistema de datos compartidos en red puede tener como máximo dos de tres propiedades deseables:

- consistencia (C), tener una única copia actualizada de los datos;
- alta disponibilidad (A) de esos datos (para actualizaciones); y
- tolerancia a particiones de red (P).

El teorema CAP prohíbe sólo una pequeña parte del espacio de diseño: la disponibilidad y consistencia perfectas en presencia de particiones.

Aunque los diseñadores aún deben elegir entre consistencia y disponibilidad cuando hay particiones presentes, existe una increíble flexibilidad para manejar las particiones y recuperarse de ellas. El objetivo moderno de CAP debería ser maximizar las combinaciones de consistencia y disponibilidad que tengan sentido para la aplicación específica. Este enfoque incluye planes para operar durante una partición y para la recuperación posterior, lo que ayuda a los diseñadores a pensar en CAP más allá de sus limitaciones históricamente percibidas.

En cierto sentido, el movimiento NoSQL se trata de crear opciones que priorizan la disponibilidad en primer lugar y la consistencia en segundo lugar; las bases de datos que siguen las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad) hacen lo contrario. ACID y BASE (Basically Available, Soft state, Eventually consistent) representan dos filosofías de diseño en los extremos opuestos del espectro de consistencia-disponibilidad. BASE PRIORIZA DISPONIBILIDAD, ACID PRIORIZA CONSISTENCIA.

Las cuatro propiedades ACID y su relación con CAP (no todas son afectadas):

- Atomicidad (A):

Todos los sistemas se benefician de operaciones atómicas. Cuando se prioriza la disponibilidad, ambas partes de una partición deben seguir utilizando operaciones atómicas. Además, las operaciones atómicas de nivel superior (del tipo que implica ACID) en realidad simplifican la recuperación.

- Consistencia (C):

En ACID, la C significa que una transacción preserva todas las reglas de la base de datos, como las claves únicas. En contraste, la C en CAP se refiere solo a la consistencia de una única copia, que es un subconjunto estricto de la consistencia ACID. Además, la consistencia ACID no puede mantenerse a través de particiones; la recuperación de particiones necesitará restaurar la consistencia ACID. En términos generales, mantener invariantes durante particiones podría ser imposible, por lo que es necesario pensar detenidamente qué operaciones prohibir y cómo restaurar invariantes durante la recuperación.

- Aislamiento (I):

El aislamiento es fundamental en el teorema CAP: si el sistema requiere aislamiento ACID, solo puede operar en la mayoría de los casos en uno de los

lados durante una partición. La serialización requiere comunicación en general y, por lo tanto, falla en particiones. Definiciones más débiles de corrección son viables en particiones mediante compensación durante la recuperación de la partición.

- Durabilidad (D):

Al igual que con la atomicidad, no hay razón para renunciar a la durabilidad, aunque el desarrollador puede optar por evitarla mediante un estado suave (en el estilo de BASE) debido a su costo. Un punto sutil es que, durante la recuperación de una partición, es posible revertir operaciones duraderas que sin saberlo violaron una invariante durante la operación. Sin embargo, en el momento de la recuperación, dada una historia duradera de ambos lados, dichas operaciones pueden detectarse y corregirse. En general, ejecutar transacciones ACID en cada lado de una partición facilita la recuperación y habilita un marco para transacciones de compensación que se pueden utilizar para la recuperación de una partición.

CAP debería permitir la consistencia perfecta (C) y la disponibilidad alta (A) la mayor parte del tiempo, pero cuando hay particiones presentes o percibidas, se necesita una estrategia que detecte las particiones y las tenga en cuenta de manera explícita. Esta estrategia debe constar de tres pasos: detectar las particiones, entrar en un modo de partición explícito que puede limitar algunas operaciones y, finalmente, iniciar un proceso de recuperación para restaurar la consistencia y compensar los errores cometidos durante una partición.

Operacionalmente, la esencia de CAP tiene lugar durante un tiempo de espera, un período en el que el programa debe tomar una decisión fundamental, la decisión de partición:

- Cancelar la operación y, por lo tanto, disminuir la disponibilidad, o
- Continuar con la operación y, por lo tanto, correr el riesgo de inconsistencia.

No existe una noción global de una partición, ya que algunos nodos pueden detectar una partición y otros no. Los nodos pueden detectar una partición y entrar en un modo de partición, que es una parte central para optimizar la consistencia (C) y la disponibilidad (A). Esta perspectiva significa que los diseñadores pueden establecer límites de tiempo de manera intencionada de acuerdo con los tiempos de respuesta deseados. Los sistemas con límites de tiempo más estrictos es probable que entren en modo de partición con más frecuencia, incluso en momentos en los que la red está simplemente lenta y no hay una partición real. ej de facebook y yahoo en el texto.

MANAGING PARTITIONS: Este enfoque de gestión consta de tres pasos:

1. Detectar el inicio de una partición.
2. Entrar en un modo de partición explícito que puede limitar algunas operaciones. La decisión sobre qué operaciones limitar depende principalmente de las invariantes que el sistema debe mantener.

3. Iniciar la recuperación de la partición cuando se restablezca la comunicación, con el objetivo de restaurar la consistencia y compensar los errores que el programa cometió mientras el sistema estaba particionado.

La mejor manera de realizar un seguimiento del historial de operaciones en ambos lados es utilizar vectores de versión, que capturan las dependencias causales entre las operaciones. Los elementos del vector son un par (nodo, tiempo lógico), con una entrada para cada nodo que ha actualizado el objeto y el momento de su última actualización. Dadas dos versiones de un objeto, A y B, A es más reciente que B si, para cada nodo en común en sus vectores, los tiempos de A son mayores o iguales que los de B y al menos uno de los tiempos de A es mayor.

Partition recovery: En algún momento, se reanuda la comunicación y la partición termina. Durante la partición, cada lado estuvo disponible y, por lo tanto, avanzó, pero la partición retrasó algunas operaciones y violó algunas invariantes. En este punto, el sistema conoce el estado y el historial de ambos lados porque mantuvo un registro cuidadoso durante el modo de partición. El estado es menos útil que el historial, a partir del cual el sistema puede deducir cuáles operaciones realmente violaron las invariantes y cuáles fueron los resultados externalizados, incluyendo las respuestas enviadas al usuario. El diseñador debe resolver dos problemas difíciles durante la recuperación:

1. El estado en ambos lados debe volverse consistente.
2. Deben existir mecanismos de compensación para los errores cometidos durante el modo de partición.

Intro relacionales (dia1)

Las 4 v de Big Data: volumen, variedad, velocidad, veracidad.

Qué es un DataBase? Una base de datos es una colección de datos almacenados permanentemente que cumple con las siguientes características:

- Lógicamente relacionados: Los datos en la base de datos están relacionados entre sí de manera lógica, lo que significa que existe una estructura y organización que conecta los datos, como tablas que se relacionan unas con otras.
- Compartidos: La base de datos puede ser accedida por múltiples usuarios o aplicaciones.
- Protegidos: El acceso a los datos en la base de datos está controlado y protegido. Esto significa que se implementan medidas de seguridad para garantizar que solo las personas autorizadas puedan acceder a ciertos datos y realizar operaciones específicas.
- Gestionados: Los datos en la base de datos se administran de manera que se mantenga su integridad y valor. Esto incluye la gestión de la consistencia de los datos, la realización de copias de seguridad y la recuperación en caso de fallas, así como la optimización del rendimiento de las consultas y transacciones.

Logical/Relational Modeling:

El Modelo Lógico:

- Debe ser el mismo independientemente del volumen de datos.
- Los datos se organizan de acuerdo a lo que representan: datos del mundo real en forma de tablas (relacionales).
- Es genérico: el modelo lógico sirve como plantilla para la implementación física en cualquier plataforma de Sistema de Gestión de Bases de Datos Relacionales (RDBMS, por sus siglas en inglés).

Normalización:

- Es el proceso de simplificar una estructura de datos compleja en una más simple y estable.
- Implica eliminar atributos redundantes, claves y relaciones del modelo de datos conceptual.

Transaction processing: es una unidad de trabajo que debe ejecutarse atómicamente y en aparente aislamiento:

1. Registro (Logging) para asegurar durabilidad.
2. Control de concurrencia (Concurrency control): Las transacciones deben aparecer como si se ejecutaran en aislamiento, lo que significa que los cambios realizados por una transacción no deben ser visibles para otras transacciones hasta que se completen. Esto se logra mediante técnicas de control de concurrencia que aseguran que las transacciones se ejecuten de manera coherente y sin interferencias mutuas.
3. Resolución de bloqueos (Deadlock resolution): En un entorno de bases de datos, las transacciones pueden competir por recursos, y esto puede llevar a situaciones de bloqueo en las que ninguna transacción puede avanzar. La resolución de bloqueos es el proceso de resolver estas situaciones de manera que las transacciones puedan continuar ejecutándose. La unidad de trabajo debe ser manejada de manera que se eviten o resuelvan los bloqueos de manera adecuada.

Relational Databases: colección de tablas lógicamente relacionadas. Las tablas de las bases de datos relacionales son un conjunto de registros. Un conjunto es una colección de elementos que no están repetidos y que no están ordenados, los datos tienen que ser distinguibles.

Primary Keys: una clave primera en una base de datos relacional es la o las columnas que identifican únicamente a una fila. Reglas para las Claves Primarias:

- Se requiere una Clave Primaria para cada tabla.
- Solo se permite una Clave Primaria en una tabla.
- Las Claves Primarias pueden constar de una o más columnas.
- Las Claves Primarias no pueden tener valores duplicados.
- Las Claves Primarias no pueden ser nulas (NULL).

- Las Claves Primarias se consideran valores "no cambiantes" o "inmutables".

Foreign Keys: son valores de una tabla que son clave primaria en otra tabla. Son opcionales. Una tabla puede tener más de una clave foránea. Una FK puede constar de más de una columna. Los valores de las FK pueden estar duplicados, pueden ser nulos (NULL) y pueden cambiar. Los valores de las Claves Foráneas (FK) deben existir en otro lugar como una Clave Primaria (PK) para mantener la integridad referencial, es decir, deben estar relacionados con una Clave Primaria en otra tabla para garantizar que los datos sean coherentes y precisos.

Las ventajas de una base de datos relacional en comparación con otras metodologías de bases de datos incluyen:

- Más flexible que otros tipos.
- Permite a las empresas responder rápidamente a condiciones cambiantes.
- Enfoque basado en datos en lugar de estar basado en aplicaciones.
- Modela el negocio, no los procesos.
- Facilita la construcción de aplicaciones, ya que los datos pueden manejar gran parte del trabajo pesado.
- Una sola copia de los datos puede servir para múltiples propósitos.
- Apoyo a la tendencia hacia la informática de usuario final.
- Fácil de entender.
- No es necesario conocer la ruta de acceso.
- Basado sólidamente en la teoría de conjuntos.

The Parsing Engine: El Motor de Análisis es responsable de:

- Gestión de sesiones individuales.
- Análisis y optimización de las solicitudes SQL.
- Generación de planes de consulta con el Optimizador. El Optimizador decide cómo se ejecutará una consulta y qué índices o métodos de acceso se utilizarán para recuperar los datos.
- Despacho del plan optimizado. Esto incluye interactuar con el Motor de Ejecución (Execution Engine) para llevar a cabo la consulta.
- Envío de la respuesta del conjunto de resultados al cliente solicitante.

DataBase Objects: Las bases de datos son repositorios para objetos de bases de datos que incluyen:

- Tablas: filas y columnas de datos. Cada fila representa una entrada o registro, y cada columna representa un atributo o campo de datos.
- Vistas: subconjuntos predefinidos de tablas existentes.
- Disparadores (Triggers): conjuntos de instrucciones SQL asociados a una tabla en particular. Se activan automáticamente cuando ocurren ciertos eventos en la tabla, como inserciones, actualizaciones o eliminaciones de registros.
- Procedimientos Almacenados (Stored Procedures): programas o rutinas que se almacenan en la base de datos y se pueden ejecutar de manera repetitiva.

- Funciones Definidas por el Usuario (User Defined Functions): programas que proporcionan funcionalidad adicional en la base de datos.

DataBase Space:

Los objetos de la base de datos requieren espacio en una base de datos o para un usuario de la siguiente manera:

- Tablas: Requieren espacio permanente (Perm Space).
- Vistas: No requieren espacio permanente (Perm Space).
- Disparadores (Triggers): No requieren espacio permanente (Perm Space).
- Procedimientos Almacenados (Stored Procedures): Requieren espacio permanente (Perm Space).
- Funciones Definidas por el Usuario (User Defined Functions - UDF): Requieren espacio permanente (Perm Space).

Los límites de espacio se especifican tanto para cada base de datos como para cada usuario y se dividen en dos categorías principales:

- Perm Space (Espacio Permanente): Es la cantidad máxima de espacio disponible para tablas permanentes y subtablas de índices en la base de datos. Este espacio se utiliza para almacenar datos a largo plazo.
- Spool Space (Espacio de Almacenamiento Temporal): Es la cantidad máxima de espacio de trabajo disponible para el procesamiento de solicitudes. Se utiliza para mantener conjuntos de resultados temporales durante el procesamiento de consultas y cálculos intermedios. Este espacio es temporal y se libera después de su uso.

Normalización (dia3)

Objetivos: preservar la información y minimizar redundancia (evitar almacenamiento de información redundante).

Pautas de Diseño: Cuatro pautas informales de diseño pueden utilizarse como medida para determinar la calidad de un diseño (estas pautas NO son siempre independientes unas de otras.):

Pauta Nro. 1. Semántica.

Estar seguro que la semántica de atributos en esquemas es clara. Diseñar esquemas tal que sea fácil de explicar su significado. No combinar atributos de diversos tipos de entidades y relaciones en una misma relación.

Pauta Nro. 2. Almacenamiento.

Reducir la información redundante en tuplas. El objetivo es minimizar espacio de almacenamiento a través del diseño. Almacenar NATURAL JOINs introduce problemas adicionales. Hay que diseñar esquemas tal que no permitan anomalías de inserción, delección y modificación; si se permiten anomalías, señalarlas claramente y asegurar que programas que actualizan BD operarán correctamente.

Notar que esta pauta puede ser violada en favor de la performance.

Pauta Nro. 3. NULLs.

Reducir la cantidad de valores NULL en tuplas. Atributos no relacionados y agrupados en una misma tabla pueden generar múltiples NULLs en una misma tupla. problemas en ppt.

Si NULLs son inevitables, asegurar que las situaciones son excepcionales y no aplican a la mayoría de las tuplas.

Pauta Nro. 4. Tuplas espúreas.

Desactivar la posibilidad de generar tuplas espúreas. Se podría perder información original. Si no está normalizada la base, un natural join puede producir tuplas espúreas, no se obtiene la información correcta.

Normalización - Formas Normales (dia3)

Dependencias Funcionales: Herramienta formal para el análisis de esquemas. Permite detectar y describir problemas descriptos previamente. Ej: $X \rightarrow Y$, entonces la variable Y depende de la variable X, "Y es funcionalmente dependiente de X".

- DF Completa: al eliminar algún atributo A de X la DF deja de existir.
- DF Parcial: es posible eliminar algún atributo A de X y la DF continúa existiendo.
- Dependencia Transitiva: existe un conjunto de atributos Z en R que no son ni Clave Candidata ni un subconjunto de alguna Clave de R, tal que $X \rightarrow Z$ y $Z \rightarrow Y$.

Normalización de los datos: Proceso de analizar los esquemas, basándose en DF y PK, con el objetivo de lograr propiedades deseables (minimizar redundancia y minimizar anomalías de inserción, delección y modificación). Esquemas que no pasan ciertos test de formas normales, se descomponen en esquemas más pequeños que pasan el test (y sus propiedades). La forma normal de una relación se refiere a la mayor forma normal alcanzada por ella.

Las formas normales, consideradas aisladas de otros factores, no garantizan un buen diseño de la BD.

Propiedades, luego de proceso de normalización por descomposición:

- Nonadditive Join (Lossless Join). Garantía de que no ocurre problema de generación de tuplas espúreas. La relación original tiene que poder ser recuperada de la descomposición. Debe lograrse a cualquier costo.
- Preservación de DF. Garantía de que cada DF se encuentra representada en algún esquema resultante de la descomposición. Es deseable, pero en algunos casos es sacrificada.

1FN:

Prohíbe: relaciones dentro de relaciones o relaciones como valores de atributos dentro de tuplas. Prohíbe relaciones anidadas.

Admite: el dominio de un atributo debe incluir sólo valores atómicos (simples e indivisibles). En la tupla, puede tomar 1 solo valor del dominio.

Técnicas para alcanzar 1FN (se pueden utilizar recursivamente para múltiples niveles):

1. Remover el atributo que viola 1FN y ubicarlo en una nueva relación. La nueva relación tiene como PK ambos atributos. Esta es la mejor opción porque no sufre de redundancia y es genérica.
2. Expandir la PK que permita que exista más de un mismo ID_ej, pero con distinta área de influencia. El problema de esta técnica es que introduce redundancia en la relación.
3. Si se conoce la máxima cantidad de valores que puede tomar el atributo, se pueden generar tantos atributos (columnas) como esa cantidad. El problema de esta técnica es que introduce NULL en casos que la tupla no posee tantos atributos.

2FN: Un esquema R está en 2FN si todo atributo no primo A de R depende funcionalmente de manera completa de la PK de R. En otras palabras, un esquema R está en 2FN si todo atributo no primo A de R no depende parcialmente (de manera funcional) de ninguna clave de R. Tip: Verificar sólo DFs cuyo lado izquierdo posea atributos que sean parte de la PK.

3FN: Un esquema R está en 3FN si está en 2FN y ningún atributo no primo de R depende transitivamente de la PK. Un esquema R está en 3FN si, para toda dependencia funcional no trivial $X \rightarrow A$ de R, se cumple alguna de las siguientes condiciones:

- X es SK de R.
- A es atributo primo de R.

NoSQL Document Databases (dia4)

Es una base no-relacional que almacena los datos como documentos estructurados. El concepto principal es el documento → Es una colección de pares: nombre de campo y valor. Los valores pueden ser un valor simple o una estructura compleja como listas, otro documento o listas de documentos hijos.

Del DER (modelo conceptual) se pasa al DID (modelo/diagrama de interrelación de documentos), y de ahí a los documentos. JSON Schema: especificación de la estructura de los documentos.

¿Cómo resolvemos la interrelación entre documentos? La decisión más importante es si incrustar o referenciar, lo que determinará el grado de desnormalización de los documentos. Referenciar: en un documento se hace referencia a un ID o una lista de ID de otro documento. Incrustar: en un documento se incluyen todos los datos (en principio) de otro documento.

¿Qué pasa cuando la cantidad de mediciones es muy grande y además se actualiza permanentemente? Se necesita crear un tipo de documento auxiliar que permita particionar las mediciones.

Wide column, Cassandra (dia4)

El modelo de datos de Cassandra se puede describir como un almacén de filas particionadas, en el cual los datos se almacenan en tablas hash multidimensionales dispersas. "Dispersas" significa que, para una fila dada, puedes tener una o más columnas, pero no es necesario que cada fila tenga todas las mismas columnas que otras filas similares (como en un modelo relacional). "Particionado" significa que cada fila tiene una clave única que permite acceder a sus datos, y las claves se utilizan para distribuir las filas en múltiples almacenes de datos.

Cassandra define una tabla como una división lógica que asocia datos similares. Una tabla de Cassandra es análoga a una tabla en el mundo relacional. Ahora no necesitamos almacenar un valor para cada columna cada vez que almacenamos una nueva entidad. Es posible que no conozcamos los valores para cada columna de una entidad determinada. En lugar de almacenar NULL para esos valores que no conocemos, lo que sería un desperdicio de espacio, simplemente no almacenaremos esa columna en absoluto para esa fila.

Cuando diseñas una tabla en una base de datos relacional tradicional, generalmente estás lidiando con "entidades" o el conjunto de atributos que describen un sustantivo en particular (hotel, usuario, producto, etc.). Normalmente no se presta mucha atención al tamaño de las filas en sí, porque el tamaño de la fila no es negociable una vez que has decidido qué sustantivo representa tu tabla. Sin embargo, cuando trabajas con Cassandra, en realidad tienes que tomar una decisión sobre el tamaño de tus filas: pueden ser anchas o delgadas, según la cantidad de columnas que contenga la fila. Una fila ancha significa una fila que tiene muchas columnas. Por lo general, hay un número menor de filas que van junto con tantas columnas. Por otro lado, podrías tener algo más parecido a un modelo relacional, donde defines un número menor de columnas y utilizas muchas filas diferentes, eso sería el modelo delgado.

Modelo Lógico: Buscar el subconjunto del modelo conceptual que satisface cada consulta. Elegir claves. Usar diagramas Chebotko para describir el modelo lógico

Reglas de Mapeo: Basados en los DMP (Data Modeling Principles), las reglas de mapeo ayudan a realizar la transición desde el modelo conceptual al modelo lógico.

MR1 (Entities and Relationships):

Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas.

MR2 (Equality Search Attributes):

Si se utilizan en una consulta por igualdad de atributos, entonces, éstos se mapean a columnas del prefijo de la clave primaria. Dichas columnas deben incluir todas las columnas de clave de partición y, opcionalmente, una o más columnas clustering key.

MR3 (Inequality Search Attributes):

Si se utilizan en consultas por desigualdad, estos atributos se mapean como columnas clustering key. En la definición de clave principal, una columna que participa en la búsqueda de desigualdad debe ubicarse después de las columnas que participan en la búsqueda de igualdad.

MR4 (Ordering Attributes):

Mapea a una columna clustering key con orden ascendente o descendente según se especifique en la consulta

MR5 (Key Attributes):

Mapea las columnas en la clave primaria. Una tabla que almacena datos de entidades o relaciones como filas debe incluir atributos claves que identifiquen estos datos únicamente.

Video Desnormalización (dia4)

Una base de datos perfectamente normalizada, puede implicar muchos join para conseguir la información necesaria.

La desnormalización toma sentido (1) durante el diseño inicial y (2) posteriormente a la implementación. (1) Representa el punto más económico para repensar y reescribir código.

Regla del 80-20 → si el 20% de los stored procedures es utilizado el 80% del tiempo, es recomendable la desnormalización. Se puede agregar clave foránea.

Por más que pueda desnormalizar, no significa que deba hacerlo:

1. ¿La desnormalización es redituable? Si no ahorra tiempo de desarrollo, tiempo de ejecución, o simplificaciones de diseño, mejor evitarla.
2. ¿Vale la pena renunciar a la tabla normalizada? Agregar columnas, índices y modificar el esquema tiene sus costos, algunos están relacionados a espacio en disco, y otros a recodificar la lógica del negocio.
3. Lo que la desnormalización ahorra en la simplificación de una consulta, podría ser costoso en el mantenimiento futuro. Las relaciones que se introduzcan en el modelo podrían no ser evidentes para aquellos que desarrollan y trabajan con la base de datos.

Desnormalizar no es una estrategia de diseño, es una forma de eludir un problema, por eso es que debe utilizarse sólo en casos de necesidad.

Si se elige desnormalizar, hay algunas reglas:

1. Desnormalizar deliberadamente. No debe ser un capricho y debe analizarse la relación costo-beneficio.
2. Documentar. El mantenimiento de un sistema desnormalizado requiere de documentación porque las relaciones no siempre son intuitivas.
3. Encapsular las modificaciones de desnormalización en transacciones.

Optimización de consultas (dia5)

Organización de Archivos se refiere a la forma en que los datos son almacenados dentro de un archivo y las formas en que pueden accederse.

Dos tipos de archivos clásicos:

- HeapFile (Registros sin orden)

Al insertarse un registro, se lo agrega al final del archivo o en alguno de los bloques con espacio libre. Las operaciones de búsqueda requieren búsqueda lineal por todos los bloques del archivo.

Ventajas: inserción. Desventajas: búsqueda y modificación.

- SortedFile (Registros ordenados a partir de una clave de búsqueda A)

Al insertarse un registro se lo agrega ordenadamente, lo que puede provocar una reorganización en los bloques del archivo. Mejoran las búsquedas por A, pero el resto de las operaciones suelen requerir una búsqueda lineal.

Índices:

Estructuras adicionales, aceleran ciertas operaciones de búsqueda sobre tablas. Mayor costo en operaciones de escritura, actualización y borrado. Mayor costo en espacio ocupado. Dos tipos de índices clásicos son:

- B+

Son árboles balanceados.

- Hash

Una tabla de hash almacena las claves de búsqueda. Cada posición de una tabla hash se asocia con un conjunto de registros. Por esta razón cada posición suele llamarse "un bucket", y los valores de hash, "índices bucket". En cada bucket se encuentra la cantidad variable de bloques.

Los índices hash solo pueden manejar comparaciones de igualdad simples. El planificador de consultas considerará el uso de un índice hash siempre que una columna indexada esté involucrada en una comparación que utilice el operador = (igual). La única ventaja de Hash es la velocidad de acceso. Pero eso depende de las colisiones.

Optimizador de consultas: Dada una consulta, encontrar un plan de ejecución eficiente.

Cuando un usuario formula una consulta, se analiza y envía esta consulta a un optimizador de consultas, que utiliza información sobre el modo que se guardan

los datos para producir un plan de ejecución eficiente para la evaluación de esa consulta. Un plan de ejecución es un detalle de las acciones que debe realizar el motor para la evaluación de la consulta, representado habitualmente como un árbol de operadores con anotaciones que contiene información detallada adicional sobre los métodos de acceso que se deben emplear, etc.

Técnicas Algebraicas → Heurísticas. Basadas en propiedades algebraicas. Árboles equivalentes. Por lo general, mejoran la performance de las consultas

Técnicas Físicas → Seleccionar implementaciones para los operadores basándose en cómo están organizados los archivos y las estructuras adicionales que existen

Utilizan el Catálogo:

- Información estadística de los datos.
- Se actualiza periódicamente y no está siempre sincronizado con los datos reales.
- Permite estimar la selectividad de los diferentes operadores.

¿Cómo se pasan los resultados entre nodos?

Materialización → Los resultados intermedios se guardan directamente en disco.

El siguiente paso de la consulta deberá levantarlos de disco nuevamente.

Pipeline → Las tuplas se van pasando al nodo superior del árbol mientras se continúa ejecutando la operación.

R \bowtie S:

Block Nested Loops Join (BNLJ): B bloques de memoria. Por cada grupo de B-2 bloques, recorro todo S. A R lo termino leyendo una sola vez en memoria.

Index Nested Loops Join (INLJ): Índice I sobre S. Por cada tupla de R, hago una búsqueda en el índice I. A R lo termino recorriendo una sola vez en memoria.

Sort Merge Join (SMJ): B bloques de memoria. Se ordenan R y S (si alguno ya está ordenado, este costo no se considera). Se hace el merge entre R y S ordenados.

Hash Join (HJ). Las tuplas de S1 se comparan solamente con las de R1. Se necesitan, para M particiones, M + 1 bloques en memoria.

Video Control de Conurrencia (dia6)

Cuando trabajamos con una base de datos que acceden muchos usuarios, vamos a trabajar con transacciones (Ti). Las transacciones pueden:

- Write [Wi(x)]: Ti escribe x.
- Read [Ri(x)]: Ti lee x.
- Commit [Ci]: consolidar los datos.
- Abort [ai]: rollback, que se olvide todo lo que sucedió en esa transacción.
- Start [S]: cuando empieza la transacción, se define con un timestamp.

La transacción es una historia serial, porque ocurre en serie. Lo ideal es que una transacción empiece y termine antes de que empiece otra, pero en la realidad muchas transacciones comienzan en simultáneo.

Ej:

$H_1 = W_1(x), R_2(x), W_1(y), R_3(y), C_2, C_3, a_1 \rightarrow$ aborto en cascada, el último aborto obliga a abortar todo.

Política optimista: dejo que hagan lo que quieran hasta que ocurra un problema, y ahí busca solución. Política pesimista: tiene una serie de reglas para que no suceda.

Vamos a ver algunos problemas de las políticas optimistas.

$H = S_1(x), S_2(x), W_2(x), R_1(x) \rightarrow$ read too late

$H = S_1(x), S_2(x), R_1(x), W_2(x) \rightarrow$ write too late

$H = s_1-w_1-s_2-r_2-a_1 \rightarrow$ dirty data \rightarrow Lee algo que no debería porque abortó

$H = s_1-s_2-w_2-w_1-c_1-c_2 \rightarrow$ regla de thomas \rightarrow se sostiene 2 porque 1 no escribió cuando debió

Estos problemas perjudican la consistencia (relación con ACID, bases de datos relacionales)

Neo4j (dia6)

Base de datos de grafos "nativa", en cuanto a la forma en que guarda y la forma en que procesa. Nombre completo: "Base de datos de grafos con propiedades de etiquetas".

Es una base de datos, no es una visualización de datos. Paradigma ACID (no BASIC), es una base de datos transaccional, es decir, que cumple con las características ACID. Así como las bases de datos relacionales guardan los datos en forma de tabla, Neo4j los guarda en forma de grafos, como un network, una red.

- Relaciones: Gráficamente corresponden a las aristas o líneas que unen los vértices. Para Neo4j esta característica genera una relación entre los puntos que conecta.
- Nodos: Son los vértices que unen las relaciones.
- Camino: conjunto de vértices interconectados por aristas.
- Propiedades: son características adicionales que se le pueden asignar tanto a los nodos como a las relaciones para otorgar información adicional y enriquecer el modelo.

Modelado de grafos

El modelado utilizado es el de grafos etiquetado, que se componen de nodos y sus relaciones.

- Los Nodos

Un nodo típicamente representa una entidad, como una persona, producto o diagnóstico de un paciente. Opcionalmente se le pueden agregar etiquetas a un nodo, indicando el rol del nodo dentro del grafo.

- Las relaciones

Para unir nodos se utilizan relaciones. Las relaciones son dirigidas, y pueden eventualmente tener propiedades asociadas. El tipo de relación provee de un predicado, mientras que la dirección de una relación muestra el sujeto y objeto.

- Etiquetas:

Los grafos llevan incorporadas etiquetas que pueden definir los distintos vértices y también las relaciones entre ellos. Por ejemplo, en Facebook podríamos tener nodos definidos por términos como 'amigo' o 'compañero de trabajo' y la relaciones como 'amigo de' o 'socio de'. Con etiquetas podemos asignar propiedades tanto a nodos como relaciones (por ejemplo, cuestiones como nombre, edad, país de residencia, nacimiento).

MOCKs:

Martes:

1. ¿Cuál es el objetivo de la normalización?

- Conseguir un diseño de la base de datos lo más pequeño posible.
- Minimizar redundancia y anomalías de inserción, borrado y modificación.
- Proteger la base de datos de potenciales ataques de seguridad.
- Proteger la integridad de los datos y hacer que la base de datos sea más flexible.

2. Dada la relación $R = \{\text{IDCiente, nombreCliente, nroCuenta, tipoCuenta, domicilio, códigoPostal}\}$ del dominio bancario ¿cuál o cuáles de las siguientes dependencias funcionales tiene sentido definir?

- códigoPostal --> domicilio
- IDCiente --> nombreCliente
- IDCiente, domicilio --> nombreCliente
- nroCuenta --> tipoCuenta

3. Dados los siguientes enunciados:

- Posteriormente a la implementación siempre conviene mantener el sistema normalizado.
- Una base de datos bien diseñada puede significar que para obtener la información necesaria haya que hacer pocos joins.
- Una base de datos normalizada puede introducir mucha complejidad en cuanto a mantenimiento.
- Una forma de desnormalizar es agregar una clave foránea a una tabla.

4. Indique qué enunciados vinculados con el Teorema CAP son verdaderos.

- Cuando ocurre una partición el programa debe decidir si cancela operaciones, aumentando así la disponibilidad.
- La disponibilidad es obviamente continua de 0 a 100 por ciento, pero también hay muchos niveles de consistencia.
- Una vez detectada una partición se debe entrar en un modo explícito de partición, limitando operaciones.
- Una vez detectada una partición se debe iniciar un proceso de recuperación para restaurar la consistencia.

5. "Las bases de datos NoSQL implementadas como sistemas distribuidos deben contar con tolerancia a las particiones que se producirán como consecuencia de interrupciones en la comunicación. Así, durante una partición, ... (marque el o los enunciados correctos).

- "...el sistema tendrá planes para la operación y para la recuperación posterior".
- "...los cajeros automáticos prohíben las extracciones de dinero".
- "...podría ocurrir que aunque la clave de una tabla es única, se permitan claves duplicadas".
- "...podría ocurrir que una computadora ejecute órdenes duplicadas".

6. Marque el enunciado correcto

- En la Primera Forma Normal todos los campos deben depender de la clave completa.
- En la Segunda Forma Normal todos los campos deben depender al menos de una parte de la clave.
- En la Tercera Forma Normal todos los campos deben depender de la clave completa y nada más que de la clave.
- Ninguno de los enunciados anteriores es correcto.

7. (Marque el enunciado correcto) "El estudio de la programación de una base de datos relacional..."

- "...consiste en el estudio de restricciones y triggers."
- "...consiste en el estudio de las restricciones especificadas."
- "...incluye el estudio de restricciones, triggers e índices."
- "...incluye el estudio de restricciones, triggers, índices y dependencias funcionales."

9. Se puede pensar en una desnormalización...

- "...como en la duplicación de columnas solicitadas con frecuencia."
- "...cuando todos los queries y stores procedures son accedidos con frecuencia homogénea."
- "...independientemente del costo en tiempo de desarrollo."
- "...separadamente del costo en tiempo de ejecución."
- "...si es menos costoso mantener la redundancia que el tiempo que le demanda al motor hacer muchos joins."

10. El administrador de una base de datos...

- ...es el único que puede ejecutar comandos de definición de datos.
- ...es quien conoce profundamente la semántica de la información persistida en la base de datos.
- ...no debería tener control sobre las sesiones de los usuarios; estos son los únicos responsables de las consultas que ejecutan.
- ...podría decidir agregar restricciones a una tabla modificando los datos acerca de los datos.

11. Dada una tabla cuyos campos son A, B, C y D, marque cuál o cuáles esquemas están en 3era forma normal. (Entendiendo por $A \rightarrow\!> B$ que "B depende funcionalmente de A" o también que "A determina a B".)

- Clave {A}. Dependencias funcionales $\{A \rightarrow B; A \rightarrow C; A \rightarrow D\}$
- Clave {A}. Dependencias funcionales $\{A \rightarrow B; A \rightarrow C; A \rightarrow D; B C y D \rightarrow A\}$
- Clave {A,B}. Dependencias funcionales $\{ A y B \rightarrow C; A y B \rightarrow D; C \rightarrow D\}$
- Clave {B,C,D}. Dependencias funcionales $\{ A \rightarrow B; A \rightarrow C; A \rightarrow D; B C y D \rightarrow A\}$

12. Marque el o los enunciados correctos

- A veces es menos costoso mantener la redundancia que el tiempo que le demanda al motor realizar múltiples joins.
- Las relaciones que se introducen en el modelo, producto de una desnormalización, pueden no ser evidentes para los usuarios de la base de datos.
- Si las columnas que duplicamos son columnas solicitadas con frecuencia no se considera que se trate de una desnormalización.
- Una forma de desnormalizar una tabla es agregarle una clave foránea.

Jueves:

1. Marque el o los enunciados correctos.

- Aunque implica un costo en espacio y operaciones, en una tabla se pueden incluir tantos índices primarios y secundarios como se desee.
- Los índices siempre aceleran las operaciones de búsqueda sobre tablas.
- Se considera buena práctica forzar al motor de la base de datos a usar un índice.
- Todos los anteriores son incorrectos.

2. Marque el enunciado correcto

- En las bases de datos relacionales el objetivo es la integración de la información.
- La existencia de software heredado no suele ser un problema a la hora de integrar o descartar bases de datos.
- Un almacén de datos realiza la operación de copiar periódicamente información desde una base de datos central hacia otras bases de datos.
- Todas las anteriores son falsas.

3. (Marque el enunciado correcto) "Block Nested Loops Join" es...

- ...un algoritmo de junta que consiste en comparar una a una las condiciones del join sin usar estructuras adicionales
- ...un operador de acceso a datos de SQL Server
- ...una estrategia de join que se resuelve completamente en memoria
- ...una operación del lenguaje SQL para hacer consultas anidadas

4. ¿Qué son las "formas normales"?

- El formato normalizado que deben tener ciertos campos en función de valores estandarizados.
- Reglas para validar si la clave de una tabla identifica únicamente los registros.
- Restricciones que deben cumplir los registros de una tabla.
- Una colección de reglas para interrelacionar tablas.

5. El término "desnormalización" se refiere a la duplicación de datos...

- ...en bases que cumplen el paradigma ACID
- ...en bases que cumplen el paradigma BASE
- ...en un sistema de computación distribuida
- ...en una columna o grupo de columnas de una tabla, que son claves primarias en otra tabla

6. (Marque el o los enunciados correctos) " En una arquitectura distribuida, al recuperarse de una partición...

- ...debe haber compensaciones por los errores producidos durante la partición."
- ...el estado de ambos lados de la partición debe volver a ser consistente."
- ...el sistema no espera a que la comunicación sea restablecida para empezar el proceso de recuperación."
- ...el sistema tiene la garantía que sigue ofreciendo disponibilidad y consistencia."

7. Los índices B+ se implementan como

- árboles balanceados
- árboles canónicos
- funciones de hash
- heap files

8. "Técnicas algebraicas" alude a

- creación de índices
- estrategias de join
- heurísticas que usa el optimizador de consultas
- sentencias SQL para actualizar el catálogo de datos

10. (Marque el o los enunciados correctos) "El lenguaje SQL...

- ...es también llamado Data Definition Language."
- ...es también llamado Data Manipulation Language."

- ...consta de subconjuntos de comandos llamados Data Definition Language y Data Manipulation Language."
- ...tiene comandos que no afectan la estructura de la base de datos y que se agrupan en el subconjunto llamado Data Manipulation Language."

11. (Marque el enunciado correcto) "Las formas normales son restricciones que deben cumplir..."

- ...las tablas de una base relacional."
- ...las claves de las tablas de una base relacional."
- ...los campos de las tablas de una base relacional."
- ...los registros de las tablas de una base relacional."



Base de datos de grafos “nativa”, en cuanto a la forma en que guarda y la forma en que procesa

Nombre completo: “Base de datos de grafos con propiedades de etiquetas”

- Es una base de datos, no es una visualización de datos.
- Paradigma ACID (no BASIC). Es un base de datos transaccional, es decir, que cumple con las características ACID.
- Así como las bases de datos relacionales guardan los datos en forma de tabla, Neo4j los guarda en forma de grafos, como un network, una red.
- Relaciones: Gráficamente corresponden a las aristas o líneas que unen los vértices. Para Neo4j esta característica genera una relación entre los puntos que conecta.
- Nodos: Son los vértices que unen las relaciones.
- Camino: conjunto de vértices interconectados por aristas.
- Propiedades: son características adicionales que se le pueden asignar tanto a los nodos como a las relaciones para otorgar información adicional y enriquecer el modelo.

Modelado de grafos

El modelado utilizado es el de grafos etiquetado, que se componen de nodos y sus relaciones.

- Los Nodos

Un nodo típicamente representa una entidad, como una persona, producto o diagnóstico de un paciente. Opcionalmente se le pueden agregar etiquetas a un nodo, indicando el rol del nodo dentro del grafo.

- Las relaciones

Para unir nodos se utilizan relaciones. Las relaciones son dirigidas, y pueden eventualmente tener propiedades asociadas. El tipo de relación provee de un predicado, mientras que la dirección de una relación muestra el sujeto y objeto.

- Etiquetas:

Los grafos llevan incorporadas etiquetas que pueden definir los distintos vértices y también las relaciones entre ellos. Por ejemplo, en Facebook podríamos tener nodos definidos por términos como ‘amigo’ o ‘compañero de trabajo’ y la relaciones como ‘amigo de’ o ‘socio de’. Con etiquetas podemos asignar propiedades tanto a nodos como relaciones (por ejemplo, cuestiones como nombre, edad, país de residencia, nacimiento).

Introducción – conceptos básicos

Cypher query sintaxis (o SQL for graphs)

Lenguaje declarativo: qué es lo que queremos, no cómo accedemos a los datos

(()) Nodes Nodo(clave:valor)

[] Relationships

- Cada nodo puede tener propiedades
- Los nodos pueden ser etiquetados con una o más etiquetas
- Las relaciones también pueden tener nombre y propiedades

(()-[]->()) Pattern

alias de un nodo **(var:)-[]->()**

(var:Process)-[]->() Process representa la etiqueta del nodo

(var:Process:Step)-[]->() un nodo se puede etiquetar con muchos nombres

(var:Process{name:'Offer'})-[]->() etiqueta con una propiedad

(()-[:HIRED]->()) la etiqueta de la relación; la relación se puede taggear con un nombre

(()-[h:HIRED]->()) etiqueta asignada al alias “h”

(()-[h:HIRED{type:'ON DEMAND'}]->()) la relación tiene una propiedad (type) y su valor es “ON DEMAND”

Comandos - queries

MATCH y **RETURN** palabras reservadas MUY UTILIZADAS

Comandos útiles

- Ctrl+space menú contextual
- Shift + enter salto de línea

//Eliminar todos los objetos de una base de datos

```
MATCH (allObjects) DETACH DELETE allObjects
```

//Crear un nodo

```
CREATE (helloWorld) RETURN helloWorld
```

//El motor crea un id pero es a fines internos, no usarlo.

//Para recuperar lo creado:

```
MATCH (allDBNodes) RETURN allDBNodes
```

//Otra forma de crear un nodo

```
CREATE (nodeVar{name: 'ciaoMondo'}) RETURN nodeVar
```

// obtengo el nodo creado

```
MATCH (nodeVar{name: 'ciaoMondo'}) RETURN nodeVar
```

//Usando la cláusula Where

```
MATCH (nodeVar)
```

```
WHERE nodeVar.name = 'ciaoMondo'
```

```
RETURN nodeVar
```

//Usando como clave de búsqueda el prefijo de un string

```
MATCH (nodeVar)
WHERE nodeVar.name STARTS WITH 'ciao'
RETURN nodeVar
```

//Que contenga, en alguna parte del nombre, el string dado

```
MATCH (nodeVar)
WHERE nodeVar.name CONTAINS 'Mondo'
RETURN nodeVar
```

//puedo crear nodos de a uno o de forma masiva

```
CREATE (city{name:'Buenos Aires'}) RETURN city
```

```
CREATE
(co {name:'Corrientes'})
,(es {name:'Escobar'})
,(ol {name:'Olivos'})
RETURN co, es, ol
```

//Parámetros: crearlos y usarlos en la misma sesión; formato json.

```
//para llamarlo voy a usar “nodes”; son tres elementos con un atributo en común (name)
:params {nodes: [{name:'Buenos Aires'}, {name:'Resistencia'}, {name:'Bolivar'}]}
```

//Comando UNWIND: permite transformar cualquier lista en listas individuales; permite iterar sobre colecciones

```
UNWIND $nodes AS nodeIt
CREATE (city{name:nodeIt.name})
RETURN city
```

```
// establecer un atributo adicional en un nodo ya creado  
MATCH (city{name: 'Buenos Aires'})  
Set city.temp= 16  
RETURN city  
// La primera vez lo crea, la posterior lo actualiza
```

//RELACIONES – parámetros

```
// sin especificar ni atributos ni nombres  
CREATE (buenosaires)-[rel:COUNTRY]->(argentina)  
RETURN buenosaires, argentina
```

```
// código para crear los nodos de ciudades con tres elementos  
:params // vamos a crear los nodos de ciudades con estos tres elementos  
cities: [  
    {  
        name:'Buenos Aires'  
        ,isCapital: true  
        ,knownFor:'La Reina del Plata'  
    }  
    ,{  
        name:'Resistencia'  
        ,isCapital: true  
        ,knownFor: ' Ciudad de las Esculturas'  
    }  
    ,{  
        name:'Corrientes'  
        , isCapital: true  
        , knownFor: 'Capital del carnaval'  
    }  
]
```

```
,{  
    name:'La Plata'  
    , isCapital: true  
    , knownFor: 'La ciudad de las diagonales'  
}  
,{  
    name:'Bogotá'  
    , isCapital: true  
    , knownFor: 'La Atenas de Sudamérica'  
}  
,{  
    name:'Escobar'  
    , isCapital: false  
    , knownFor: 'Capital Nacional de la flor'  
}  
]
```

```
//idem para países  
:params  
countries: [  
    {  
        name:'Argentina'  
    },  
    {  
        name:'Colombia'  
    }  
]
```

Tengo dos parámetros: voy a crear nodos con la ayuda de los parámetros creados

UNWIND \$countries AS countryIterator

CREATE (countryVar:Country)

SET countryVar=properties(countryIterator)

RETURN countryVar

//De la misma forma creo las ciudades mutatis mutandi

UNWIND \$cities AS citiesIterator

CREATE (cityVar:City)

SET cityVar=properties(citiesIterator)

RETURN cityVar

//Creando relaciones

MATCH (city {name:'Buenos Aires'}) , (country {name:'Argentina'})

CREATE (city) -[relVar :COUNTRY]-> (country)

RETURN city, country