



Martin Kleppmann



Dgsm

is also good for the environment.
efficiency. In October 2009, V
Variety of food, including P
Or that's what you might know
your music library now.

young
Various
changing, but the
softly
practice

Grade Level	Percentage (%)
6th	10
7th	15
8th	18
9th	22
10th	25
11th	28
12th	30

- Peer number, location information, and other details
 - hosted by individuals or organizations
 - Individuals and organizations involved in living arrangements
 - Witnesses of crime
 - Neighbors, relatives, and other family members
 - Police officers and other law enforcement officials
 - Friends, neighbors, and other individuals who may have been present during the incident
 - Modem, telephone, and other electronic devices
 - Previous communication or interaction from the witness

Create a free account

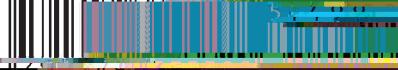
Marshall University
Cambridge, UK. Tel: +44 1296 383255; fax: +44 1296 383255;
internet: <http://www.marshall.ac.uk/~mccormic>; e-mail: mccormic@marshall.ac.uk
largest collection of multiple choice questions in English language teaching
and other subjects.

DARAJAH



ANSWER

ISBN: 978-3-7434-4422-1



10 of 10

Designing Data-Intensive Applications

*The Big Ideas Behind Reliable, Scalable,
and Maintainable Systems*

Martin Kleppmann



Technology is a powerful force in our society. Data, software, and communication can be used for bad: to entrench unfair power structures, to undermine human rights, and to protect vested interests. But they can also be used for good: to make underrepresented people's voices heard, to create opportunities for everyone, and to avert disasters. This book is dedicated to everyone working toward the good.

Table of Contents

Preface.....	xiii
--------------	------

Part I. Foundations of Data Systems

1. Reliable, Scalable, and Maintainable Applications.....	3
Thinking About Data Systems	4
Reliability	6
Hardware Faults	7
Software Errors	8

Relational Versus Document Databases Today	38
Query Languages for Data	
Declarative Queries on the Web	44

12. The Future of Data Systems.....	489
Data Integration	490
Combining Specialized Tools by Deriving Data	490
Batch and Stream Processing	494
Unbundling Databases	499
Composing Data Storage Technologies	499
Designing Applications Around Dataflow	504
Observing Derived State	509
Aiming for Correctness	515
The End-to-End Argument for Databases	516
Enforcing Constraints	521
Timeliness and Integrity	524
Trust, but Verify	528
Doing the Right Thing	533
Predictive Analytics	533
Privacy and Tracking	536
Summary	543
Glossary.....	553
Index.....	559

Preface

If you have worked in software engineering in recent years, especially in server-side and backend systems, you have probably been bombarded with a plethora of buzz words relating to storage and processing of data. NoSQL! Big Data! Web-scale! Sharding! Em.55o 60lvnsistency! ACID! CAP theorem! Cloud services! MapReduce! Real-time!

which it is changing—as opposed to colas69-intensive , where CPU cycles are the bottleneck.

The tools and technologies that help data-intensive applications store and process



This book is for software engineers, software architects, and technical managers who love to code. It is especially relevant if you need to make decisions about the architecture of the systems you work on—for example, if you need to choose tools for solving a given problem and figure out how best to apply them. But even if you have no choice over your tools, this book will help you better understand their strengths and weaknesses.

You should have some experience building web-based applications or network services, and you should be familiar with relational databases and SQL. Any non-relational databases and other data-related tools you know are a bonus, but not required. A general understanding of common network protocols like TCP and HTTP is helpful. Your choice of programming language or framework makes no difference for this book.

If any of the following are true for you, you'll find this book valuable:

- You want to learn how to make data systems scalable, for example, to support web or mobile apps with millions of users.
- You need to make applications highly available (minimizing downtime) and operationally robust.
- You are looking for ways to maintain the system, even as they grow and as requirements and technologies change.
- You have a natural curiosity for the way things work and want to know what goes on inside major websites and online services. This book breaks down the internal processes various databases and data structures, and it's great fun to explore the bright parts that went into their design.

Sometimes, when discussing reliable data systems, people make assumptions and conclusions



How to Contact Us

Please address comments and questions concerning this book to the publisher:

OJ260Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/designing-data-intensive-apps>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

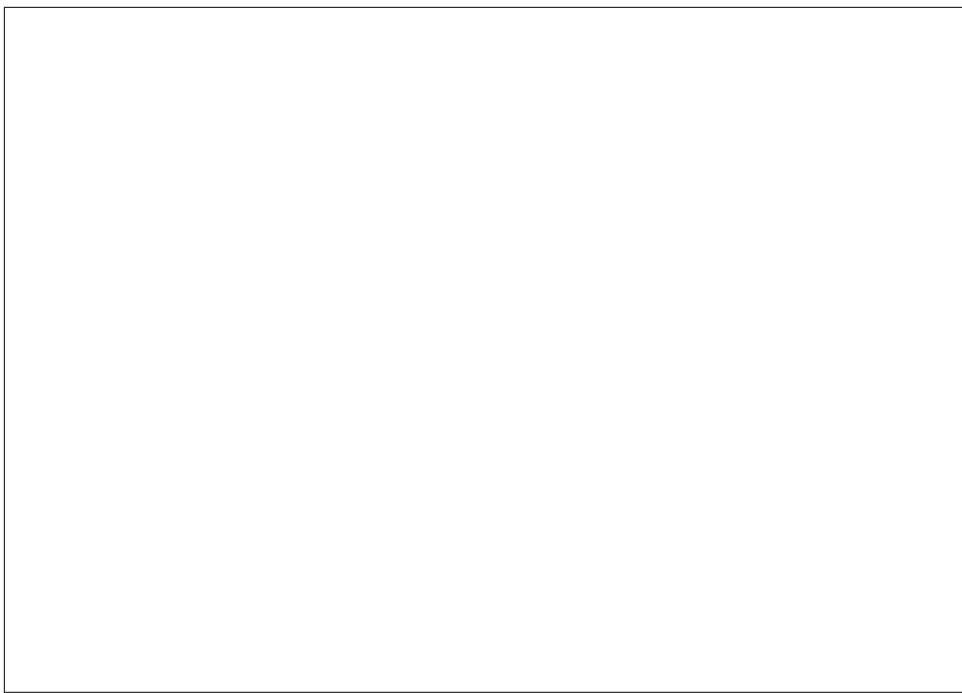
Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter:

McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O’Neil, Onora O’Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, and Brett Wooldridge.

Several more people have been invaluable to the writing of this book by reviewing





In this book, we focus on three concerns that are important in most software systems:

Reliability

space—good luck getting that budget item approved. So it only makes sense to talk about tolerating *certain types* of faults.

Note that a fault is not the same as a failure [2]. A fault is usually defined one com k



- A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.
-



median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

This makes the median a good metric if you want to know how long users typically have to wait: half of user requests are served in less than the median response time, and the other half take longer than the median. The median is also known as the *50th*



While distributing stateless services across multiple machines is fairly straightforward, taking stateful data systems from a single node to a distributed setup can intro

- Anticipating future problems and solving them before they occur (e.g., capacity planning)
- Establishing good practices and tools for deployment, configuration management, and more
-



formance. In a scalable system, you can add processing capacity in order to remain reliable under high load.

Maintainability has many facets, but in essence it's about making life better for the

[24] Jeffrey Dean and Luiz André Barroso: “[The Tail at Scale](#),” *Communications of the ACM*, volulrTj , number 2, pages 74–80, February 2013. doi: [10.1145/2408776.2408794](https://doi.org/10.1145/2408776.2408794)

[25]



groups of people—for example, the engineers at the database vendor and the applica

application code and the database model of tables, rows, and columns. The disconnect between the models is sometimes called an *impedance mismatch*.ⁱ

i. A term borrowed from electronics. Every electric circuit has a certain impedance (resistance to alternating current) on its inputs and outputs. When you connect one circuit's output to another one's input, the power transfer across the connection is maximized if the output and input impedances of the two circuits match. An impedance mismatch can lead to signal reflections and other troubles.

Many-to-One and Many-to-Many Relationships

In [Example 2-1](#) in the preceding section,

ii. Literature on the relational model distinguishes several different normal forms, but the distinctions are of little practical interest. As a rule of thumb, if you're duplicating values that could be stored in just one place, the schema is not normalized.



-
- iii. At the time of writing, joins are supported in RethinkDB, not supported in MongoDB, and only supported in predeclared views in CouchDB.
-

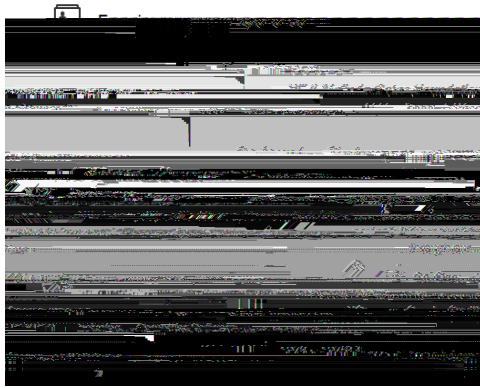


Figure 2-3. The company name is not just a string, but a link to a company entity.
Screenshot of linkedin.com.

Figure 2-4 illustrates how these new features require many-to-many relationships. The data within each dotted rectangle can be grouped into one document, but the references to organizations, schools, and other q21-s need to be represented as references, and require joins when queried.

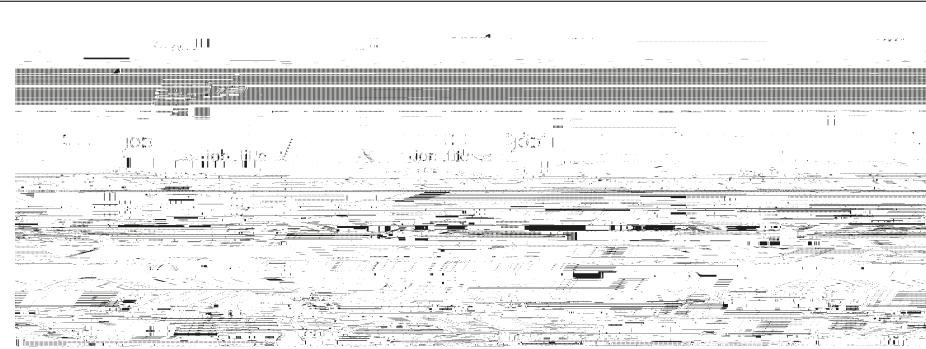


Figure 2-4. Extending résumés with many-to-many relationships.

Are Document Databases Repeating History?

While many-to-many relationships and joins are routinely used in relational databases, document databases and NoSQL reopened the debate on how best to represent such relationships in a database. This debate is much older than NoSQL—in fact, it goes back to the very earliest computerized database systems.

The most popular database for business data processing in the 1970s was IBM's *Information Management System* (IMS), originally developed for stock-keeping in the

iv. Foreign key constraints allow you to restrict modifications, but such constraints are not required by the relational mode⁹. Even with constraints, joins on foreign keys are performed at query time, whereas in

the query optimizer, not by the application developer, so we rarely need to think about them.



where σ (the Greek letter sigma) is the selection operator, returning only those animals that match the condition $family = "Sharks"$.

Declarative Queries on the Web



If you were using XSL instead of CSS, you could do something similar:

```
<xsl:template match="li[@class='selected']/p">
  <fo:block background-color="blue">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Here, the XPath expression `li[@class='selected']/p` is equivalent to the CSS selector `li.selected > p` in the previous example. What CSS and XSL have in common is that they are both *declarative* languages for specifying the styling of an element.

Imagine what life would be like if you had to use an imperative approach. In JavaScript, using the core Document Object Model (DOM) API, the result might look something like this:

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < el =
```

The `map` function would be called once for each document, resulting in `emit("1995-12", 3)` and `emit("1995-12", 4)`. Subsequently, the `reduce` function would be called with `reduce("1995-12", [3, 4])`, returning 7.

The `map` and `reduce` functions are somewhat restricted in what they are allowed to do. They must be *pure* functions, which means they only use the data that is passed to them as input, they cannot perform additional database queries, and they must not have any side effects. These restrictions allow the database to run the functions anywhere, in any order, and rerun them on failure. However, they are nevertheless powerful: they can parse strings.

MapReduce is a fairly low-level programming model for distributed execution on a cluster of machines. Higher-level query languages like SQL can be implemented as a pipeline of MapReduce operations (see [Chapter 10](#)), but there are also many distributed implementations of SQL that don't use MapReduce. Note there is nothing in SQL that constrains it to running on a single machine, and MapReduce doesn't have a monopoly on distributed query execution.

Being able to use JavaScript code in the middle of a query is a great feature for advanced queries, but it's not limited to MapReduce—some SQL databases can be extended with JavaScript functions too [34].

A usability problem with MapReduce is that you have to write two carefully coordinated JavaScript functions, which is often harder than writing a single query. Moreover, a declarative query language offers more opportunities for a query optimizer to improve the performance of a query. For these reasons, MongoDB has chosen to support both approaches.

Graph-Like Data Models

We saw earlier that many-to-many relationships are an important distinguishing feature between different data models. If your application has mostly one-to-many relationships (tree-structured data) or no relationships between records, the document model is appropriate.

But what if many-to-many relationships are very common in your data? The relational model can handle simple cases of many-to-many relationships, but as the connections within your data become more complex, it becomes more natural to start modeling your data as a graph.

A graph consists of two kinds of objects: *vertices* (also known as *nodes* or *entities*) and *edges* (also known as *relationships* or *arcs*). Many kinds of data can be modeled as a graph. Typical examples include:

Social graphs

Vertices are people, and edges indicate which people know each other.

The web graph

Vertices are web pages, and edges indicate HTML links to other pages.

Road or rail networks

Vertices are junctions, and edges represent the roads or railway lines between them.

tusdrs;edges indicate which people kre jfriesd oithimach other.,which pahecindhanp265

- The vertex at which the edge ends (the *head vertex*)
- A label to describe the kind of relationship between the two vertices
- A collection of properties (key-value pairs)

You can think of a graph store as consisting of two relational tables, one for vertices



intricacies of sovereign states and nations), and varying granularity of data (Lucy's current residence is specified as a city, whereas her place of birth is specified only at the level of a state).

You could imagine extending the graph to also include many other facts about Lucy and Alain, or other people. For instance, you could use it to indicate any food aller

matches any two vertices that are related by an edge labeled BORN_IN. The tail vertex of that edge is bound to the variable person, and the head vertex is left unnamed.

Example 2-4. Cypher query to find people who emigrated from the US to Europe

```
MATCH
  (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us: Location {name: 'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu: Location {name: 'Europe'})
RETURN person.name
```

The query can be read as follows:

Find any vertex (call it person) that meets *both* of the following conditions:

1. person has an outgoing BORN_IN edge to a Location vertex with name United States

-- *lives_in_europe* is the set of vertex IDs of all people living in Europe
lives_in_europe(vertex_id

vii. Technically, Datomic uses 5-tuples rather than triples; the two additional fields are metadata for version

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model [43]. (It is an acronym for *SPARQL Protocol and RDF Query Language*, pronounced “sparkle.”) It

viii. Datomic and Cascalog use a Clojure S-expression syntax for Datalog. In the following examples we use a

DNA molecule) and matching it against a large database of strings that are similar, but not identical. None of the databases described here can handle this kind of usage, which is why researchers have written specialized genome database

- [10] “RethinkDB 1.11 Documentation,” *rethinkdb.com*, 2013.
- [11] “Apache CouchDB 1.6 Documentation,” *docs.couchdb.org*, 2014.
- [12] Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: “On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform,” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[13][13

[]

“13



[43] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux: “**SPARQL 1.1 Query Language**,” W3C Recommendation, March 2013.

[44] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou: “**Data log and Recursive Query Processing**,” *Foundations and Trends in Databases*, volume



CHAPTER 3

Storage and Retrieval



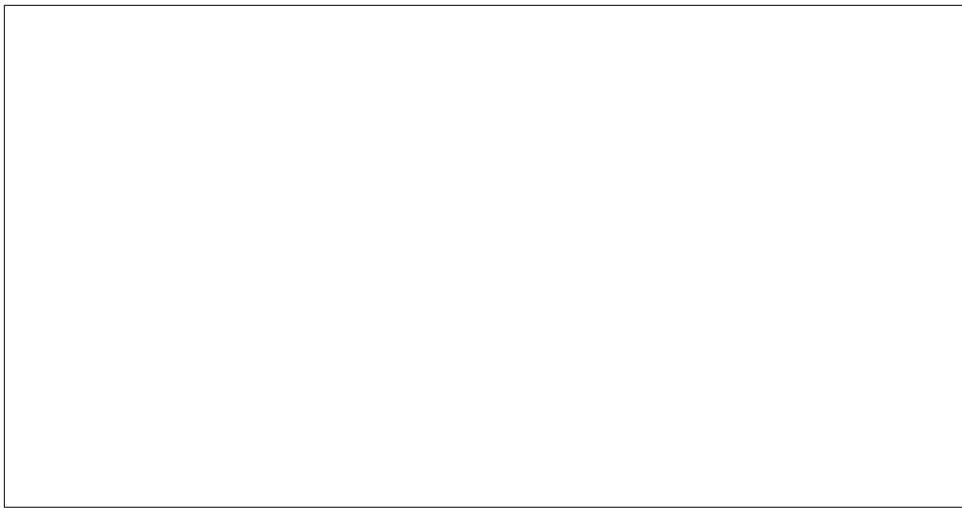


Figure 3-3. Performing compaction and segment merging simultaneously.

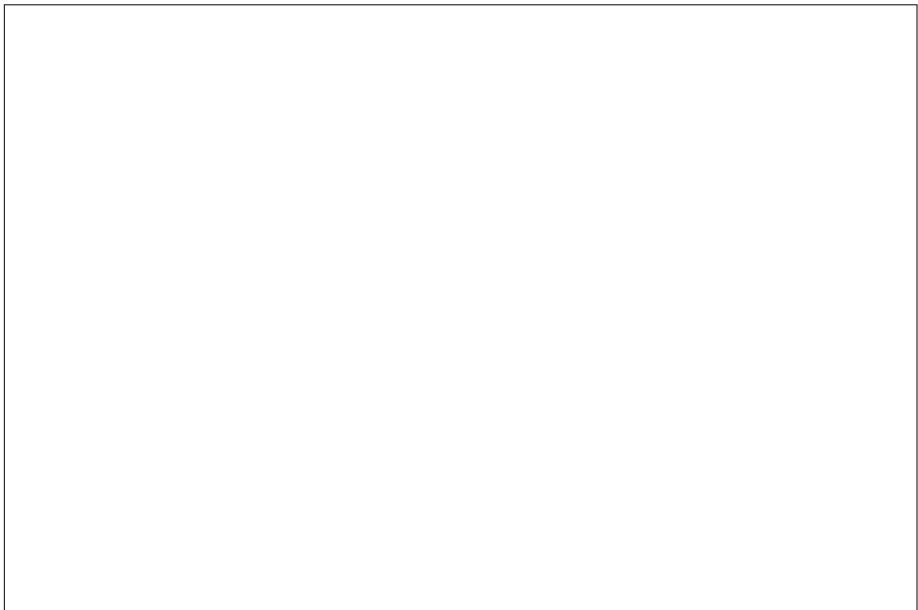
Each segment now has its own in-memory hash table, mapping keys to file offsets. In order to find the value for a key, we first check the most recent segment's hash map;



a snapshot of each segment's hash map on disk, which can be loaded into memory more quickly.

Partially written records





Constructing and maintaining SSTables

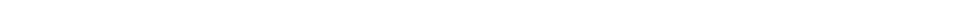
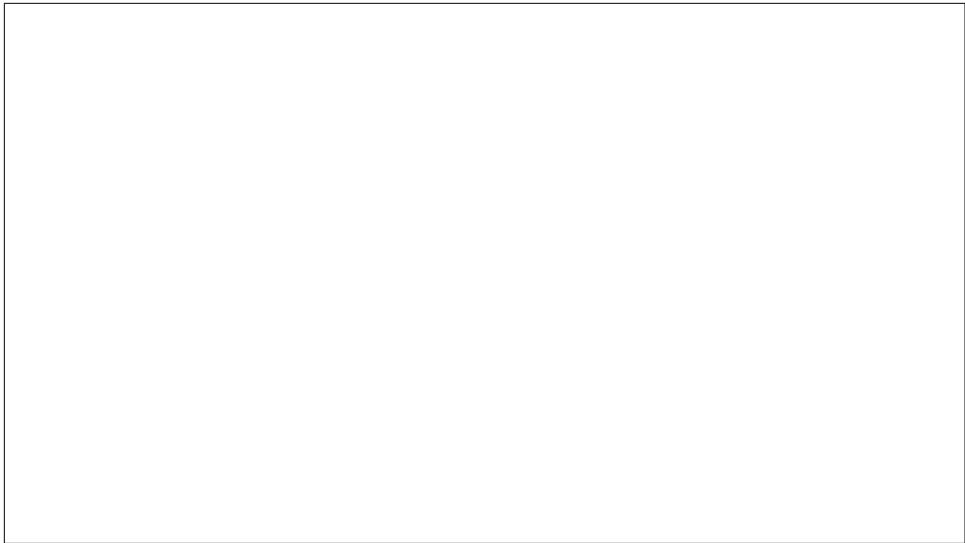
Fine so far—but how do you get your data to be sorted by key in the first place? Our incoming writes can occur in any order.

Maintaining a sorted structure on disk is possible (see “[B-Trees” on page 79](#)), but maintaining it in memory is much easier. There are plenty of well-known tree data structures that you can use, such as red-blackslt in [\[19\]](#) VLkslt in [\[r\]](#)

log-structured filesystems [11]. Storage engines that are based on this principle of merging and compacting sorted files are often called LSM storage engines.

Lucene, an indexing engine for full-text search used by Elasticsearch and Solr, uses a similar method for storing its *term dictionary* [12, 13]. A full-text index is much more

Introduced in 1970 [17] and called “ubiquitous” less than 10 years later [



Eventually we get down to a page containing individual keys (a *leaf page*), which

the ithe

B-treeawan



ii. Inserting a new key into a B-tree is reasonably intuitive, but deleting one (while keeping the tree balanced) is somewhat more involved [2].

Making B-trees reliable

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page;



rency control, as we shall see in “[Snapshot Isolation and Repeatable Read](#)” on [page 237](#).

- We can save space in pages by not storing the entire key, but abbreviating it.

iii. This variant is sometimes known as a B⁺ tree, although the optimization is so common that it often isn't distinguished from other B-tree variants.

Advantages of LSM-trees

As with any kind of duplication of data, clustered and covering indexes can speed up reads, but they require additional storage and can add overhead on writes. Databases also need to go to additional effort to enforce transactional guarantees, because applications should not see inconsistencies due to the duplication.







In the rest of this chapter we will look at storage engines that are optimized for analytics instead.

The divergence between OLTP databases and data warehouses

The data model of a data warehouse is most commonly relational, because SQL is generally a good fit for analytic queries. There are many graphical data analysis tools that generate SQL queries, visualize the results, and allow analysts to explore the data (through operations such as *drill-down* and *slicing and dicing*)

the total sales per store yesterday, you just need to look at the totals along the appropriate dimension—no need to scan millions of rows.

Finishing off the OLTP side, we did a brief tour through some more complicated indexing structures, and databasea 1.nt are optimized for keeping all data in m 6tk.d

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: “[Bigtable: A Distributed Storage System for Structured Data](#),” at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

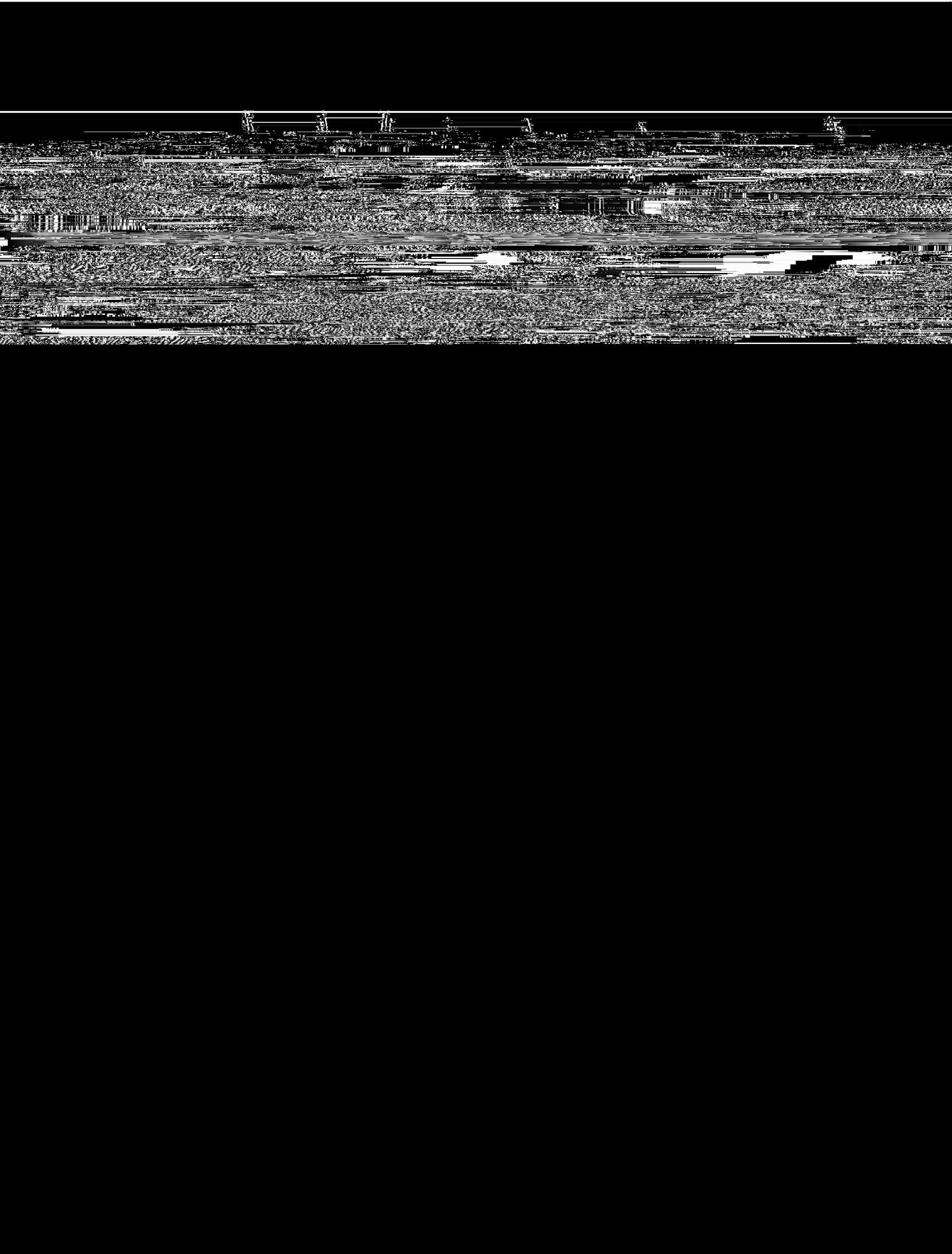
[10] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil: “[The Log-Structured Merge-Tree \(LSM-Tree\)](#),” *Acta Informatica*, volume 33, number 4, pages 351–385, June 1996. doi:[10.1007/s002360050048](https://doi.org/10.1007/s002360050048)

[11]



[

[54] Sergey Melnik, Andrey Gubarev, Jing Jing Long, et al: “



CHAPTER 4

Encoding and Evolution

Everything changes and nothing stands still.

—Heraclitus of Ephesus, as quoted by Plato in *Cratylus* (360 BCE)

Applications inevitably change over time. Features are added or modified as new products are launched, user requirements become better understood, or business circumstances change. In Chapter 1 we quoted the idea of *evolvability*. We should

- Versioning data is often an afterthought in these libraries: as they are intended for quick and easy encoding of data, they often neglect the inconvenient problems of forward and backward compatibility.
- Efficiency (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought. For example, Java's built-in serialization is notorious for its bad performance and bloated encoding [



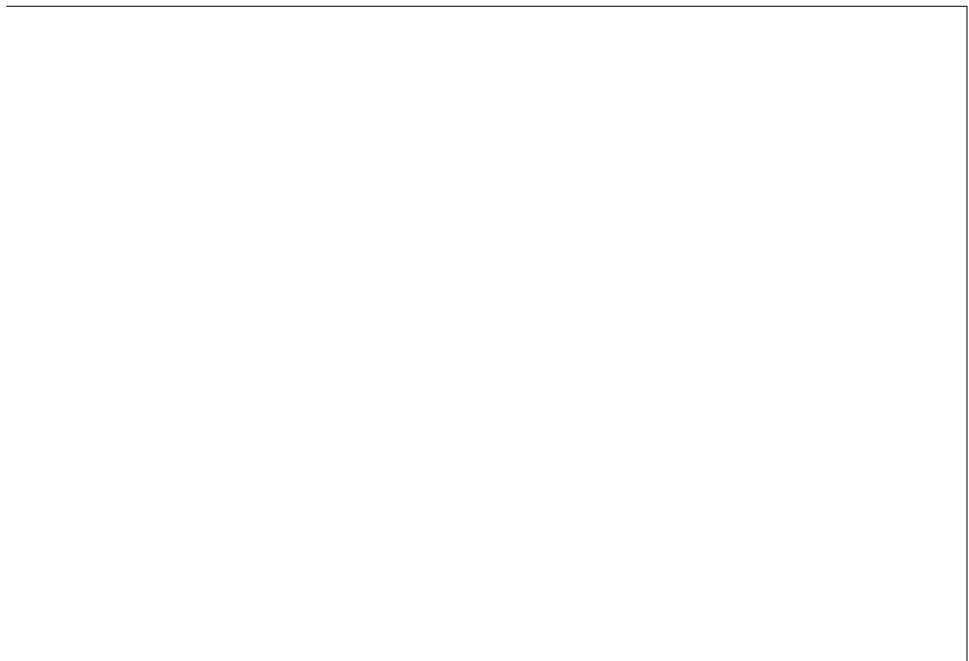
- There is optional schema support for both XML [11] and JSON [12]. These schema languages are quite powerful, and thus quite complicated to learn and implement. Use of XML schemas is fairly widespread, but many JSON-based tools don't bother using schemas. Since the correct interpretation of data (such

Example 4-1. Example record which we will encode in several binary formats in this chapter

```
{  
    "userName": "Martin",  
    "favoriteNumber": 1337,  
    "interests": ["daydreaming", "hacking"]  
}
```

Let's look at an example of MessagePack, a binary encoding for JSON. [Figure 4-1](#) shows the byte sequence that you get if you encode the JSON document in [Example 4-1](#) with MessagePack [14]. The first few bytes are as follows:

1. The first byte, 0x83, indicates that what follows is an object (top four bits =0x80) with three fields (bottom four bits = 0x03). (In case you're wondering what happens if an object has more than 15 fields, so that the number of fields doesn't fit in four bits, it then gets a different type indicator, and the number of fields is encoded in tw5inder of fields is





In some programming languages, `null` is an acceptable default for any variable, but

iv. To be precise, the default value must be of the type of the *first* branch of the union, although this is a specific limitation of Avro, not a general feature of union types.

tags.) This kind of dynamically generated schema simply wasn't a design goal of



That's a fairly abstract idea—there are many ways data can flow from one process to another. Who encodes the data, and who decodes it? In the rest of this chapter we

The problems with remote procedure calls (RPCs)

Web services are merely the latest incarnation of a long line of technologies for making API requests over a network, many of which received a lot of hype but have serious problems. Enterprise JavaBeans (EJB) and Java's Remote Method Invocation (RMI) are limited to Java. The Distributed Component Object Model (DCOM) is limited to Microsoft platforms. The Common Object Request Broker Architecture (CORBA) is excessively complex, and does not provide backward or forward compatibility [41].

All of these are based on the idea of a *remote procedure call* (RPC), which has been around since the 1970s [42]. The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called *location transparency*). Although RPC seems convenient at first, the approach is fundamentally flawed [43, 44]. A network request is very different from a local function call:

- A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control. A network request is unpredictable: the request or response may be lost due to a network problem, or the remote machine may be slow or unavailable, and such problems are entirely outside of your control. Network problems are common, so you have to anticipate them, for example by retrying a failed request.
- A local function call either returns a result, or throws an exception, or never returns (because it goes into an infinite loop or the process crashes). A network request has another possible outcome: it may return without a result, due to a *timeout*. In that case, you simply don't know what happened: if you don't get a response from the remote service, you have no way of knowing whether the request got through or not. (We discuss this issue in more detail, in [Chapter 8](#))
- If you retry a failed network request, it could happen that the requests are actually getting through, and only the responses are getting lost. In that case, `retryingLocati(a`

For these reasons, REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organi

- *Akka*
-

vague about datatypes, so you have to be careful with things like numbers and

[45] Marius Eriksen: “Your Server as a Function,” at

machines, you don't need to be operating at Google scale: even for small companies,

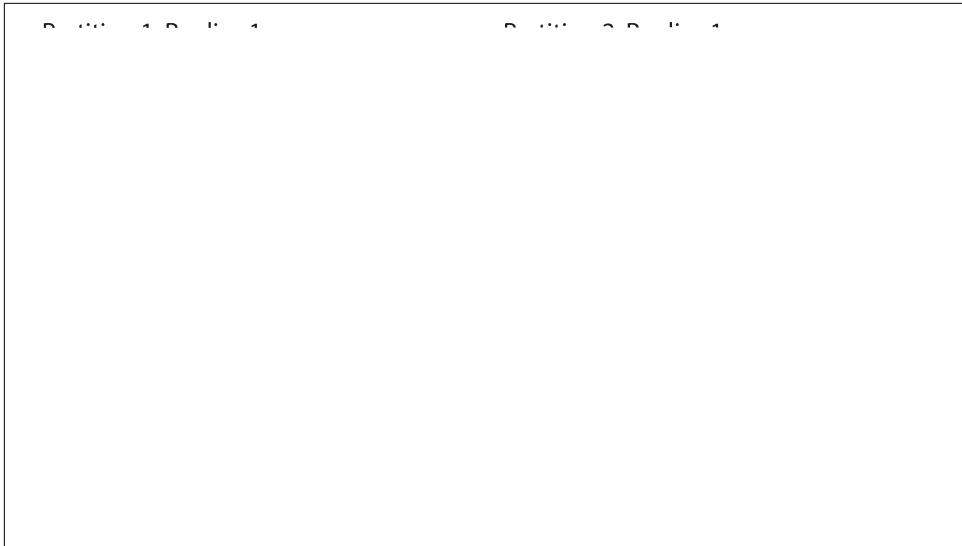


Figure II-1. A database split into two partitions, with two replicas per partition.

With an understanding of these concepts, we can discuss the difficult trade-offs that you need to make in a distributed system. We'll discuss *transactions* in [Chapter 7](#), as that will help you understand all the many things that can go wrong in a data system, and what you can do about them. We'll conclude this part of the book by discussing the fundamental limitations of distributed systems in [Chapters 8 and 9](#).

Later, in [Part III](#) of this book, we will discuss how you can take several (potentially distributed) datastores and integrate them into a larger system, satisfying the needs of a complex application. But first, let's talk about distributed data.

References

- [1] Ulrich Drepper: “[What Every Programmer Should Know About Memory](#),” *akka dia.org*, November 21, 2007.
- [2] Ben Stopford: “[Shared Nothing vs. Shared Disk Architectures: An Independent View](#),” *benstopford.com*, November 24, 2009.
- [3] Michael Stonebraker: “[The Case for Shared Nothing](#),” *IEEE Database Engineering Bulletin*, volume 9, number 1, pages 4–9, March 1986.
- [4] Frank McSherry, Michael Isard, and Derek G. Murray: “[Scalability! But at What COST?](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

CHAPTER 5

Replication

3.



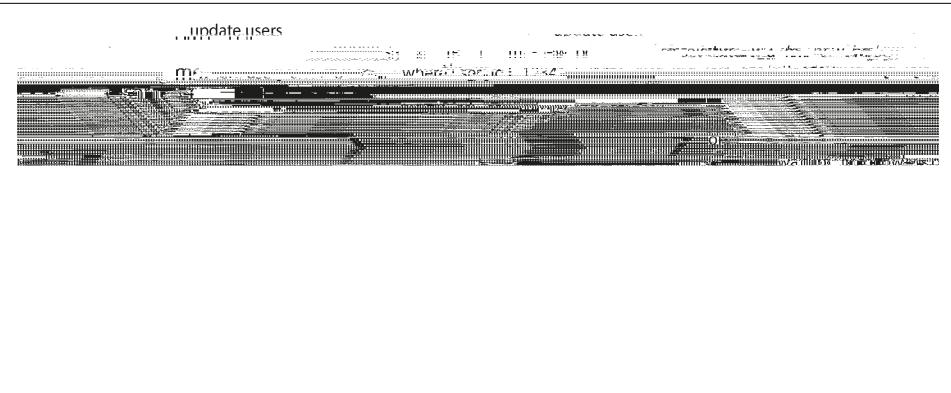


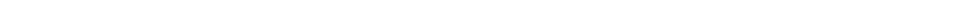
Figure 5-2. Leader-based replication in memcached one synchronous and one asynchronous follower.

In the example of Figure 5-2, the replication to follower 1 is *synchronous*: the leader waits until follower 1 has confirmed that it received the write before reporting success.

follower 2 is asynchronousfollower 2 implements fnormallysetoffollower 2 is to the user, akeatind

leader and one synchronous follower. This configuration is sometimes also called *semi-synchronous* [7].

Often, leader-based replication is configured to be completely asynchronous. In this case, if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client. However, a fully asynchronous configura].



Leader failure: Failover

Handling a failure of the leader is trickier: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called *failover*.

Failover can happen manually (an administrator is notified that the leader has failed and takes the necessary steps to make a new leader) or automatically. An automatic failover process usually consists of the following steps:

- 1.



rows, but because the new leader's counter lagged behind the old leader's, it reused some primary keys that were previously assigned by the old leader. These primary keys were also used in a Redis store, so the reuse of primary keys resulted in inconsistency between MySQL and Redis, which caused some private data

ii. This approach is known as *fencing* or, more emphatically, *SHOOTDOWN* other Node In The Head. I discuss fencing in more detail in “[The leader and the lock](#)” on page 301.

When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.

This method of replication is used in PostgreSQL and Oracle, among others [16]. The main disadvantage is that the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. This makes replication closely coupled to the storage engine. If the database changes its storage format Tw 01.11053 Tw

iii. The term *eventual consistency*



Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

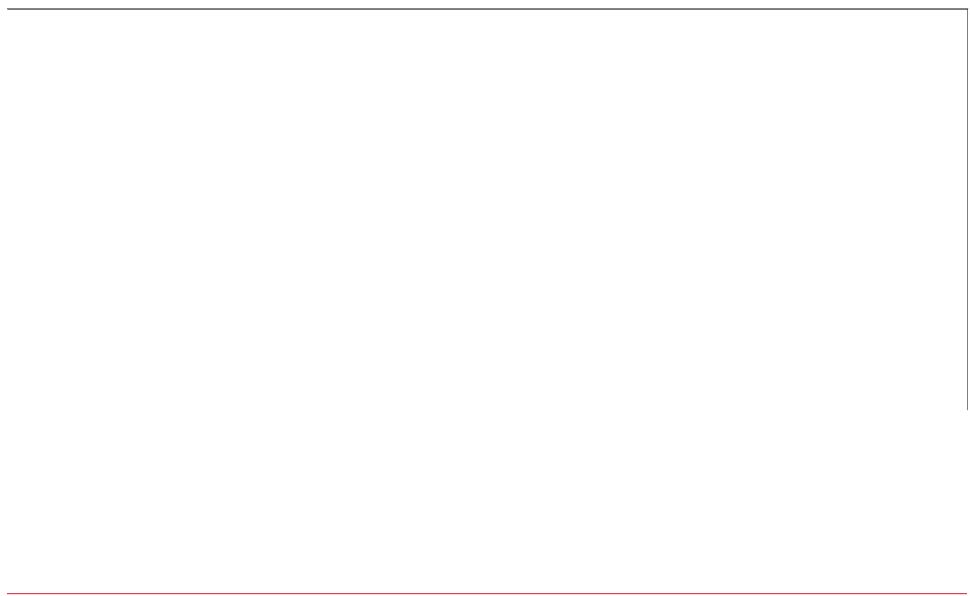
In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency* [24]. This is a guarantee that if the user reloads the page, they will always see any updates they submitted themselves. It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly.

How can we implement read-after-write consistency in a system with leader-based replication? There are various possible techniques. To mention a few:

- ntee ETQBT /F3 10.5 Tf 2.18437 Tw 1 0 0 -1 17.997 300.27356 Tm(When reading something that th

Some databases support multi-leader configurations by default, but it is also often implemented with external tools, such as Tungsten Replicator for MySQL [26], BDR for PostgreSQL [27], and GoldenGate for Oracle [19].

Although multi-leader replication has advantages, it also has a big downside: the same data may be concurrently modified in two different datacenters, and those write



Synchronous versus asynchronous conflict detection

In a single-leader database, the second writer will either block and wait for the first

convergent way, which means that all replicas must arrive at the same final value



Automatic Conflict Resolution

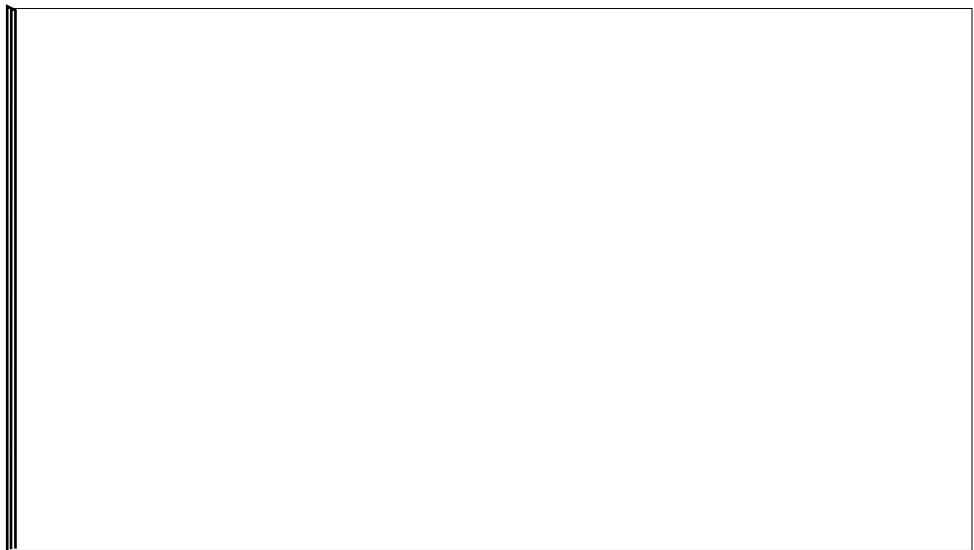
Multi-Leader Replication Topology

In a multi-leader replication topology, multiple nodes can be active leaders simultaneously. This can lead to conflicts if two nodes try to update the same data at the same time.



v. Not to be confused with a *star schema* (see “[Stars and Snowflakes: Schemas for Analytics](#)” on page 93), which describes the structure of a data model, not the communication topology between nodes.

with its own identifier, that data change is ignored, because the node knows that it has already been processed.



every write is not sufficient, because clocks cannot be trusted to be sufficiently in sync

vi. Dynamo is not available to users outside of Amazon. Confusingly, AWS offers a hosted database product called *DynamoDB*, which uses a completely different architecture: it is base9 0 using single-leader application.

Two mechanisms are often used in Dynamo-style datastores:

Read repair

When a client makes a read from several nodes in parallel, it can detect any stale responses. For example, in [Figure 5-10](#), user 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stale value and writes the newer value back to that replica. This approach works well for values that are frequently read.

vii. Sometimes this kind of quorum is called a *strict quorum*, to contrast with *sloppy quorums* (discussed in “[Sloppy Quorums and Hinted Handoff](#)” on page 183).

If fewer than the required w or r nodes are available, writes or reads return an error. A node could be unavailable for many reasons: because the node is down (crashed, powered down), due to an error executing the operation (can't write because the disk is full), due to a network interruption between the client and the node, or for any number of other reasons. We only care whether the node returned a successful response and don't need to distinguish between different kinds of fault.

Limitations of Quorum Consistency

If you have n replicas, and you choose w and r such that $w + r > n$, you can generally expect every read to return the most recent value written for a key. This is the case because the set of nodes to which you've written and the set of nodes from which you've read must overlap. That is, among the nodes you read there must be at least one node with the latest value (illustrated in [Figure 5-11](#))

- If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it's undetermined whether the read returns the old or the new value.
- If a write succeeded on some replicas but failed on others (for example because the disks on some nodes are full), and overall succeeded on fewer than w replicas, it is not rolled back on the replicas where it succeeded. This means that if a write was reported as failed, subsequent reads may or may not return the value from that write [47].
- If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below w , breaking the quorum condition.
- Even if everything is working correctly, there are edge cases in which you can get unlucky with the timing, as we shall see in “[Linearizability and quorums](#)” on page 334.

Thus, although quorums appear to guarantee that a read returns the latest written value, in practice it is not so simple. Dynamo-style databases are generally optimized for use cases that can tolerate eventual consistency. The parameters w and r allow you to trade off consistency guarantees against availability and latency.

47and 47rEverubs0452 15loal),7.99) absolute0438guaranteesTd(.) J Tw ubsIhater- snedsd valueeaki2.6 Td

called *hinted handoff*. (Once you find the keys to your house again, your neighbor politely asks you to get off their couch and go home.)

Sloppy quorums are particularly useful for increasing write availability: as long as *any* w nodes are available, the database can accept writes. However, this means that even when $w + r > n$, you cannot be sure to read the latest value for a key, because the

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.
- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge tog 34erl4o ss.og 34erl4o ss.oeandnom the prist re. Tm(T65)Tj 10684 0 Tw





Summary

[21] Greg Sabino Mullane: “[Version 5 of Bucardo Database Replication System](#),”

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: “

[53] Jonathan Ellis: “Why Does the Pope Not Like Meddling? Cardinal Muller and Matt Walsh: [Search of the]

CHAPTER 6

Partitioning



-
- i. Partitioning, as discussed in this chapter, is a way of intentionally breaking a large database down into smaller ones. It has nothing to do with *network partitions* (netsplits), a type of fault in the network between nodes. We will discuss such faults in [Chapter 8](#).
-

However, the downside of key range partitioning is that certain access patterns can lead to hot spots. If the key is a timestamp, then the partitions correspond to ranges of time—e.g., one partition per day. Unfortunately, because we write data from the

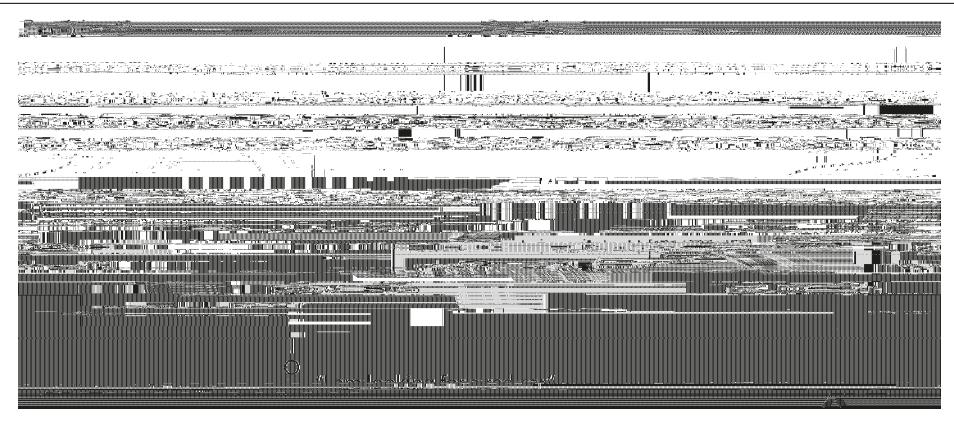


ii. If your database only supports a key-value model, you might be tempted to implement a secondary index



Figure 6-4. Partitioning secondary indexes by document.

In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition. It



partitions of the index (every term in the document might be on a different partition, on a different node).

In an ideal world, the index would always be up to date, and every document written to the database would immediately be reflected in the index. However, in a term-partitioned index, that would require a distributed transaction across all partitions affected by a write, which is not supported in all databases (see [Chapter 7](#) and [Chapter 9](#)).

In practice, updates to global secondary indexes are often asynchronous (that is, if you read the index shortly after a write, the change you just made may not yet be

Dynamic partitioning

For databases that use key range partitioning (see “Partitioning by Key Range” on page 202



ber of partitions, the size of each partition is proportional to the size of the dataset. In both of these cases, the number of partitions is independent of the number of nodes.

For example, LinkedIn's Espresso uses Helix [31]

The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load). This requires choosing a partitioning scheme that is appropriate to your data, and rebalancing the partitions when nodes are added to or removed from the cluster.

We discussed two main approaches to partitioning:

- *Key range partitioning*, where keys are sorted, and a partition owns all the keys from some minimum up to some maximum. Sorting has the advantage that efficient range queries are possible, but there is a risk of hot spots if the application often accesses keys that are close together in the sorted order.

In this approach, partitions are typically rebalanced dynamically by splitting the range into two subranges when a partition gets too big.

- *Hash partitioning*, where a hash function is applied to e0.5643u4ih5two subran1235



to several partitions can be difficult to reason about: for example, what happens if the write to one partition succeeds, but another fails? We will address that question in the following chapters.

References

- [1] David J. DeWitt and Jim N. Gray: “**Parallel Database Systems: The Future of High Performance Database Systems**,” *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992. doi:10.1145/129888.129894
- [2] Lars George: “**HBase vs. BigTable Comparison**,” *larsgeorge.com*, November 2009.
- [3] “**The Apache HBase Reference Guide**,” Apache Software Foundation, *hbase.apache.org*
- [4] MongoDB, Inc.: “**New Hash-Based Sharding Feature in MongoDB 2.4**,” *blog.mongodb.org*, April 10, 2013.
- [5] Ikai Lan: “**App Engine Datastore Tip: Monotonically Increasing Values Are Bad**,” *ikaisays.com*, January 25, 2011.
- [6] Martin Kleppmann: “**Java’s hashCode Is Not Safe for Distributed S**

[16] Richard Low: “[The Sweet Spot for Cassandra Secondary Indexing](#),” *wentnet.com*

CHAPTER 7

Transactions

For decades, *transactions*

In the late 2000s, nonrelational (NoSQL) databases started gaining popularity. They aimed to improve upon the relational status quo by offering a choice of new data models (see [Chapter 2](#)), and by including replication ([Chapter 5](#)) and partitioning ([Chapter 6](#)) by default. Transactions were the main casualty of this movement: many of this new generation of databases abandoned transactions entirely, or redefined the word to describe a much weaker set of guarantees than had previously been understood [4].

With the hype around this new crop of distributed databases, there emerged a popular belief that transactions were the antithesis of scalability, and that any large-scale system would have to abandon transactions in order to maintain good performance and high availability [5, 6]. On the other hand, transactional guarantees are sometimes presented by database vendors as an essential requirement for “serious applications” with “valuable data.” Both viewpoints are pure hyperbole.

ing. For example, in multi-threaded programming, if one thread executes an atomic operation, that means there is no way that another thread could see the half-finished result of the operation. The system can only be in the state it was before the operation

Replication and Durability

Single-Object and Multi-Object Operations

To recap, in ACID, atomicity and isolation describe what the database should do if a client makes several writes within the same transaction:

ii. Arguably, an incorrect counter in an email application is not a particularly critical problem. Alternatively, think of a customer account balance instead of an unread counter, and a payment transaction instead of an email.

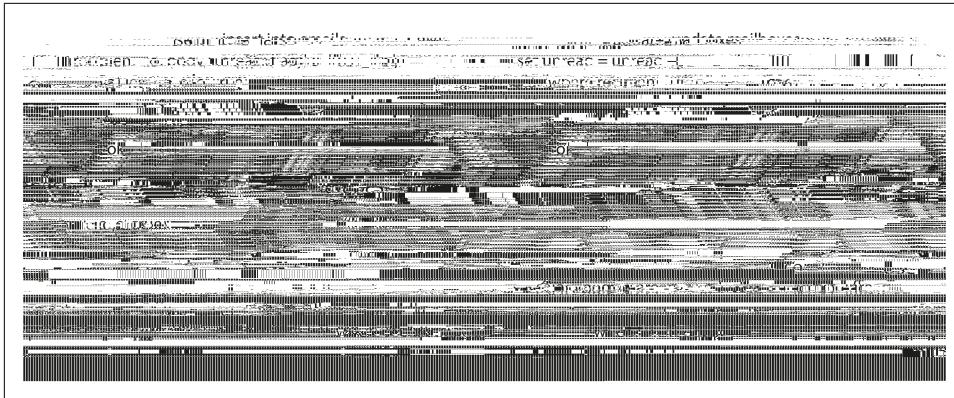


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a “dirty read”).

Figure 7-3 illustrates the need for atomicity: if an error occurs somewhere over the course of the transaction, the contents of the mailbox and the unread counter might become out of sync. In an atomic transaction, if the update to the counter fails, the counter fails,s(0001 d(co) Tj 0 g the)Tn atomic trd update t 17lthe b th..0029 0 45 5.25 cm q q 25 436.880



iii. This is not ideal. If the TCP connection is interrupted, the transaction must be aborted. If the interruption happens after the client has requested a commit but before the server acknowledges that the commit happened, the client doesn't know whether the transaction was committed or not. To solve this issue, a transaction manager can group operations by a unique transaction identifier that is not bound to a particular TCP connection. We will return to this topic in “[The End-to-End Argument for Databases](#)” on page 516.

On the other hand, many nonrelational databases don't have such a way of grouping operations together. Even if there is a multi-object API (for example, a key-value store may have a *multi-put*

iv. Strictly speaking, the term *atomic increment* uses the word *atomic* in the sense of multi-threaded programming. In the context of ACID, it should actually be called *isolated* or *serializable* increment. But that's getting nitpicky.





Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

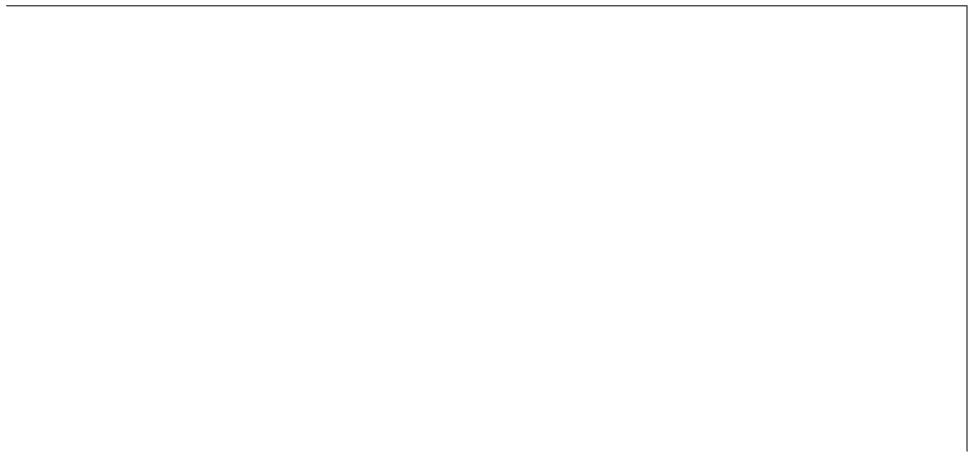
Implementing read committed

Read committed is a very popular isolation level. It is the default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, and many other databases [8].

Most commonly, databases prevent dirty writes by using row-level locks: when a transaction wants to modify a particular object (row or document), it must first acquire a lock on that object. It must then hold that lock until the transaction is committed or aborted. Only one transaction can hold the lock for any given object; if another transaction wants to write to the same object, it must wait until the first transaction is committed or aborted before it can acquire the lock and continue. This locking is done automatically by databases in read committed mode (or stronger isolation levels).

How do we prevent dirty reads? One option would be to use the same lock, and to require any transaction that wants to read an object to briefly acquire the lock and then release it again immediately after reading. This would ensure that a read couldn't happen while an object has a dirty, uncommitted value (because during that time the lock would be held by the transaction that has made the write).





incoming payment has arrived (with a balance of \$500), and the other account after the outgoing transfer has been made (the new balance being \$400). To Alice it now appears as though she only has a total of \$900 in her accounts—it seems that \$100 has vanished into thin air.



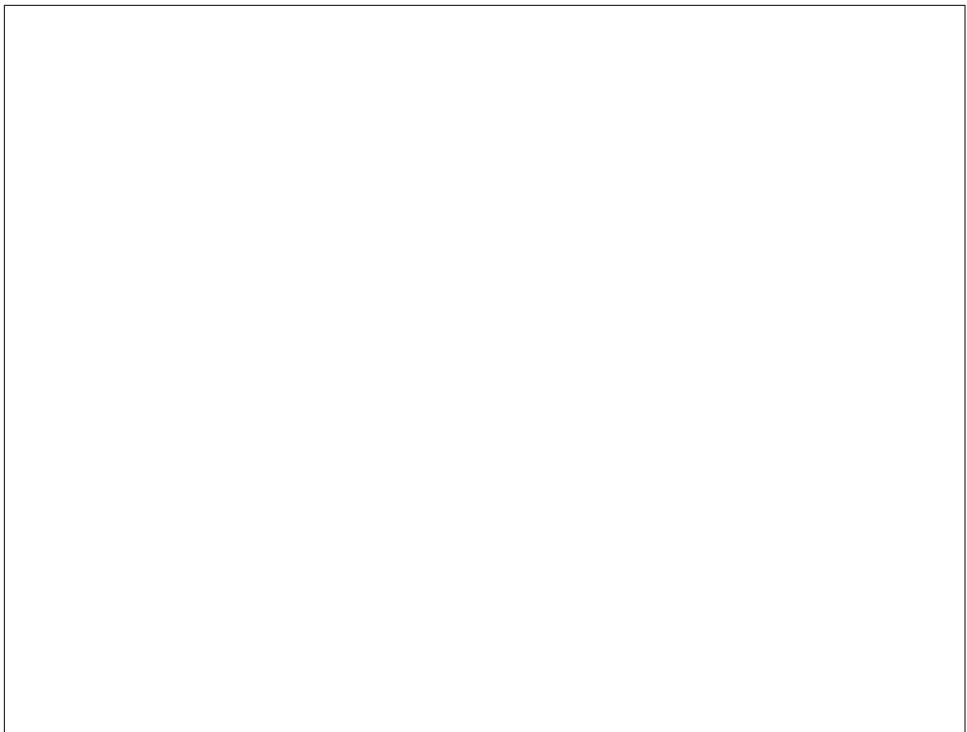


Figure 7-7. Implementing snapshot isolation using multi-version objects.



applied. This technique is sometimes known as *cursor stability* [36, 37]. Another option is to simply force all atomic operations to be executed on a single thread.

Unfortunately, object-relational mapping frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [38]. That's not a problem if you know what you are doing, but it is potentially a source of subtle bugs that are difficult to find by testing.

Explicit locking

Another option for preventing lost updates, if the database's built-in atomic operations don't provide the necessary functionality, is for the application to explicitly lock objects that are going to be updated. Then the application can perform a read-modify-write cycle, and if any other transaction tries to concurrently read the same object, it is forced to wait until the first read-modify-write cycle has completed.

For example, consider a multiplayea game in which several playeas can move the same figure concurrently. In this case, an atomic operation may not be sufficient, because the application also needs to ensure that a playea's move abides by the rules of the game, which involves some logic that you cannot sensibly implement as a data base query. Instead, you may use a lock to prevent two playeas from concurrently moving the same piece, as illustrated in [Example 7-1](#).

[Example 7-1 Explicitly locking rows to prevent lost updates](#) OTd() TJ O/F53333 rg 4.25 OTdfi gures) TnTw -6

doctor would have been prevented from going off call. The anomalous behavior was only possible because the transactions ran concurrently.

You can think of write skew as a generalization of the lost update problem. Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects). In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).

We saw that there are various different ways of preventing lost updates. With write skew, our options are more restricted:

-

In the case of the doctor on call example, the row being modified in step 3 was one of the rows returned in step 1, so we could make the transaction safe and avoid write skew by locking the rows in step 1 (SELECT FOR UPDATE). However, the other four examples are different: they check for the *absence* of rows matching some search condition, and the write *adds* a row matching the same condition. If the query in step 1 doesn't return any rows, SELECT FOR UPDATE can't attach locks to anything.

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a *phantom* [3]. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the examples we discussed, phantoms can lead to particularly tricky cases of write skew.

Materializing conflicts

If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?

Even though this seems like an obvious idea, database designers only fairly recently—

Pros and cons of stored procedures

-
- x. If a transaction needs to access data that's not in memory, the best solution may be to abort the transac
-

queue may form if several transactions want to access the same object, so a transaction may have to wait for several others to complete before it can do anything.

For this reason, databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles (see “[Describing Performance](#)” on page 13) if there is contention in the workload. It may take just one slow transaction, or one transaction that accesses a lot of data and acquires many locks, to cause the rest of the

- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.

The key idea here is that a predicate lock applies even to objects that do not yet exist



action is aborted and has to be retried. Only transactions that executed serializably

In



The rate of aborts significantly affects the overall performance of SSI. For example, a transaction that reads and writes data over a long period of time is likely to run into conflicts and abort, so SSI requires that read-write transactions be fairly short (long-

References

- [1] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al.: “**A History and Evaluation of System R**,” *Communications of the ACM*, volume 24, number 10, pages 632–646, October 1981. doi:[10.1145/358769.358784](https://doi.org/10.1145/358769.358784)
- [2] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger:

[12] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge: “**Understanding the**

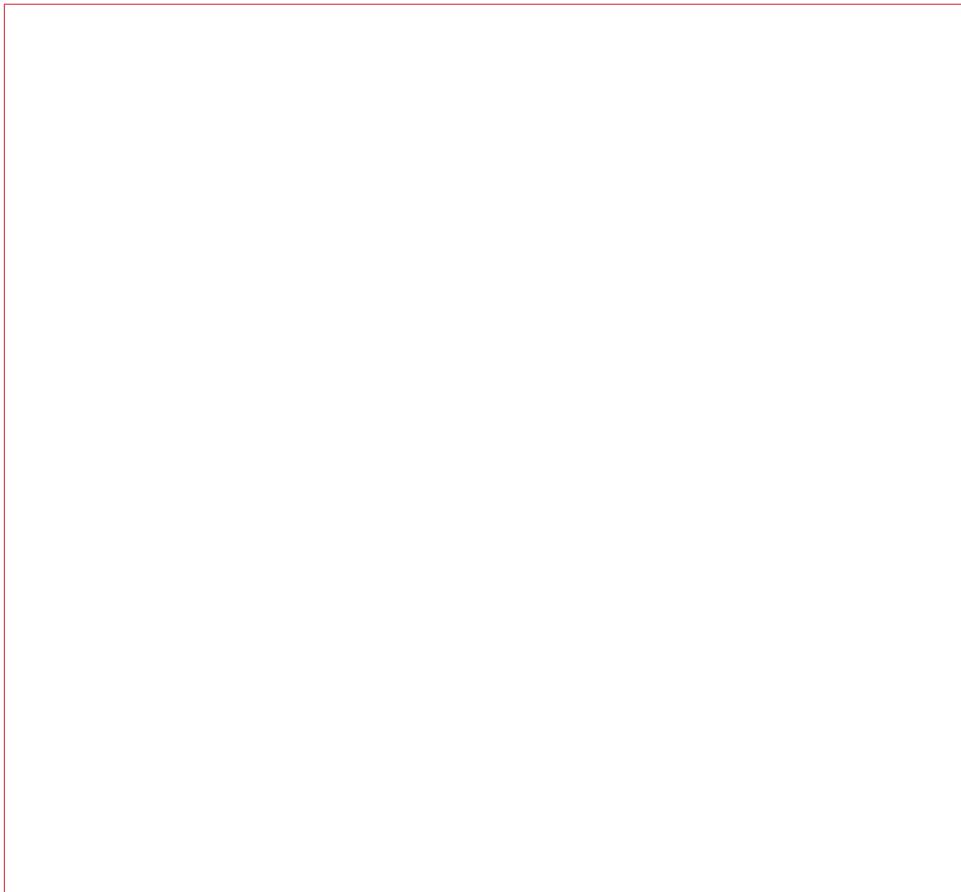
[28] Hal Berenson, Philip A. Bernstein, Jim N. Gray, et al.: “[A Critique of ANSI SQL Isolation Levels](#),” at

Symposium on Operating Systems Principles (SOSP), December 1995. doi:
[10.1145/224056.224070](https://doi.org/10.1145/224056.224070)

[44] Gary Fredericks: “



will



The sender can't even tell whether the packet was delivered: the only option is for the



Even if network faults are rare in our environment, the fact that faults *can* occur means that our software needs to be able to handle them. Whenever any communication happens over a network, it may fail—there is no way around it.

If the error handling of network faults is not defined and tested, arbitrarily bad things could happen: for example, the cluster could become deadlocked and permanently unable to serve requests, even when the network recovers [20], or it could even delete all of our data [21].

- If you have access to the management interface of the network switches in your datacenter, you can query them to detect link failures at a hardware level (e.g., if



This kind of network is *synchronous*: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been reserved in the next hop of the network. And because there is no queueing, the maximum end-to-end delay is 60ms. This is because 16 bits, which is equivalent to 4 bytes, can be transmitted in 4 microseconds. Therefore, the maximum end-to-end delay is 4 microseconds.

From the above analysis, we can conclude that the maximum end-to-end delay is 4 microseconds.

ii. Except perhaps for an occasional keepalive packet, if TCP keepalive is enabled.

iii. *Asynchronous Transfer Mode* (ATM) was a competitor to Ethernet in the 1980s [32], but it didn't gain much adoption outside of telephone network core switches. It has nothing to do with automatic teller machines (also known as cash machines), despite sharing an acronym. Perhaps, in some parallel universe, the internet is based on something like ATM—in that universe, internet video calls are probably a lot more reliable than they are in ours, because they don't suffer from dropped and delayed packets.

v. Although the clock is called *real-time*

wrong. If your NTP daemon is misconfigured, or a firewall is blocking NTP traffic, the clock error due to drift can quickly become large.



needed in order to prevent violations of causality (see “[Detecting Concurrent Writes](#)” on page 184).

- It is possible for two nodes to independently generate writes with the same time



doesn't know any more precisely than that [58]. If we only know the time $+/- 100$ ms, the microsecond digits in the timestamp are essentially meaningless.



the lease before it expires. If the node fails, it stops renewing the lease, so another node can take over when it expires.

You can imagine the request-handling loop looking something like this:

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

What's wrong with this code? Firstly, it's relying on synchronized clocks: the expiry time on the lease is set by a different machine (where the expiry may be calculated as the current time plus 30 seconds, for example), and it's being compared to the local system clock. If the clocks are out of sync by more than a few seconds, this code will start doing strange things.

Secondly, even if we change the protocol to only use the local monotonic clock, there



reduced by changing allocation patterns or tuning GC settings [66], we must assume the worst if we want to offer robust guarantees.

- In virtualized environments, a virtual machine can be *suspended* (pausing the

resumed
process can occur at any time in a process's execution) and can last for an arbitrary long time. This feature is sometimes used for *live migration* of virtual

length53426 Twi

All of these occurrences can *preempt* the running thread at any point and resume it at some later time, without the thread even noticing. The problem is similar to making multi-threaded code on a single machine thread-safe: you can't assume anything about timing, because arbitrary context switches and parallelism may occur.

When writing multi-threaded code on a single machine, we have 7a sdj3good tools for making it thread-safe: mutexes, semaphores, atomic counters, lock-free data struc



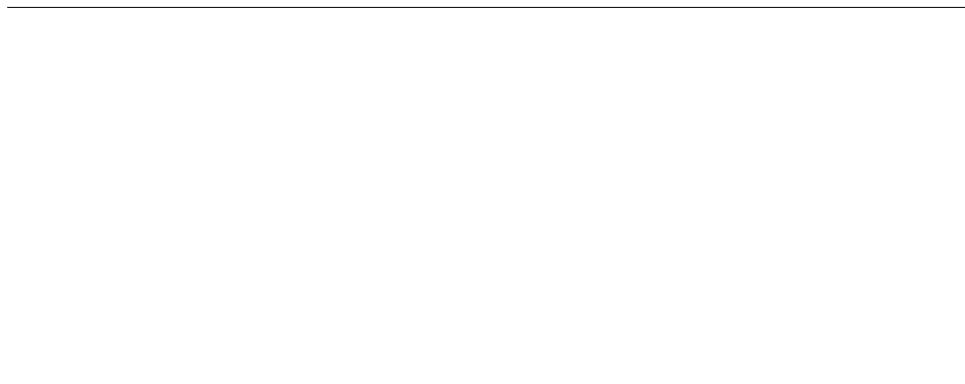
Knowledge, Truth, and Lies

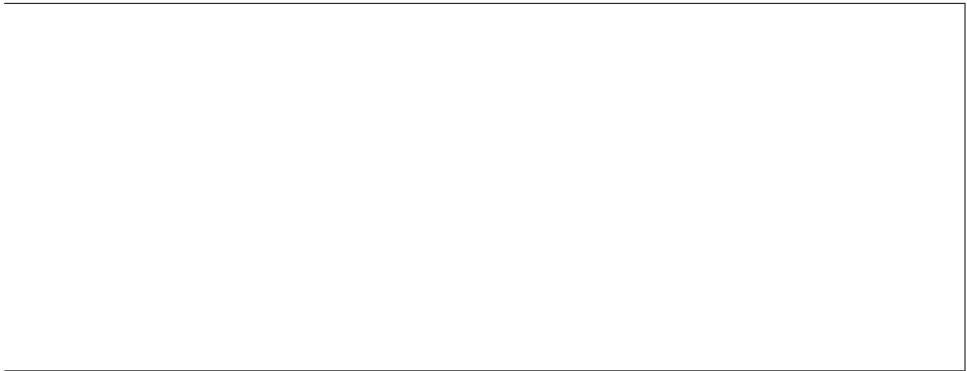
So far in this chapter we have explored the ways in which distributed systems are different from programs running on a single computer: there is no shared memory, only



In a slightly less nightmarish scenario, the semi-disconnected node may notice that the messages it is sending are not being acknowledged by other nodes, and so realize that there must be a fault in the network. Nevertheless, the node is wrongly declared dead by the other nodes, and the semi-disconnected node cannot do anything about it.

As a third scenario, imagine a node that experiences a long stop-the-world garbage collection pause. All of the node's threads are preempted by the GC and paused for one minute, and consequently, no requests are processed and no responses are sent. The other nodes wait, retry, grow impatient, and eventually declare the node dead





A system is *Byzantine fault-tolerant* if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attack is made on the network. This concern is relevant in certain specific circumstances, for example, in aerospace environments, the data in a computer's memory or CPU register can be corrupted by a physical failure or a malicious attack. Extending fault tolerance to such environments will be very expensive.

Similarly, it would be appealing if a protocol could protect us from vulnerabilities, security compromises, and malicious attacks. Unfortunately, this is not realistic either: in most systems, if an attacker can compromise one node, they can probably

turn requires that we somehow formalize the kinds of faults that we expect to happen in a system. We do this by defining a *system model*, which is an abstraction that describes what things an algorithm may assume.

With regard to timing assumptions, three system models are in common use:

Synchronous model

The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error. This does not imply exactly synchronized clocks or zero network delay; it just means you know that network delay, pauses, and j 0.0 drift will never exceed some fixed upper bound [88]. The synchronous model is not a realistic model of most practical systems, because (as discussed in this chapter) unbounded delays and pauses do occur.

Partially synchronous model

Partial synchrony means that a system behaves like a sydels *most of the time*, but it sometimes exceeds the bounds for network delay, process pauses, and j 0.0 drift [88]. This is a realistic model of many systems: most of the time, networks and processes are quite well behaved—otherwise we would never be able to get anything done—but we have to reckon with the fact that any timing assumptions may be shattered occasionally. When this happens, network delay, pauses, and j 0.0 error may become arbitrarily large.

Asynchronous model

In this model, an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a j 0.0 (so it cannot use timeouts). Some algorithms can be designed for the asynchronous model, but it is very restrictive.

Moreover, besides timing issues, we have to consider node failures. The three most common system models for nodes are:

Crash-stop faults

In the crash-stop model, an algorithm may assume that a node can fail in only one way, namely by crashing. This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever—it never comes back.

Crash-recoto cofaults

We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time. In the crash-recoto comodel, nodes are assumed to have stable storage (i.e., nonvolatile disk storage) that is present across Tj 0 Tw -0.000

For modeling real systems, the partially synchronous model with crash-recovery

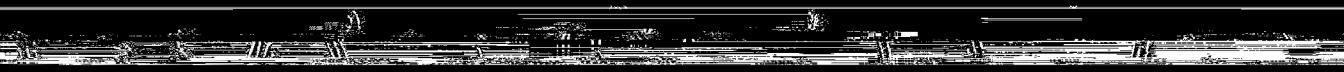
theory. (In practice, if a bad configuration change is rolled out to all nodes, that will

[41] James C. Corbett, Jeffrey Dean, Michael Epstein, et al.:

[56] Leslie Lamport: “**Time, Clocks, and the Ordering of Events in a Distributed System**,” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978.

[70] David Terei and Amit Levy: “**Blade: A Data Center Garbage Collector**

[86] Jonathan Stone and Craig Partridge: “



However, this is a very weak guarantee—it doesn't say anything about *when* the repli



Linearizability

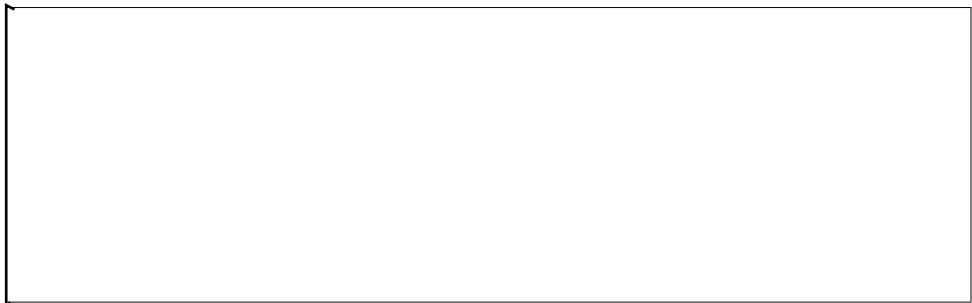
In an eventually consistent database, if you ask two different replicas the same question at the same time, you may get two different answers. That's confusing. Wouldn't it be a lot simpler if the database could give the illusion that there is only one replica (i.e., only one copy of the data)? Then every client would have the same view of the data, and you wouldn't have to worry about replication lag.

This is the idea behind *linearizability*

Figure 9-1 shows an example of a nonlinearizable sports website [9]. Alice and Bob are sitting in the same room, both checking their phones to see the outcome of the 2014 FIFA World Cup final. Just after the final score is announced, Alice refreshes the page, sees the winner announced, and excitedly tells Bob about it. Bob incredulously hits *reload* on his own phone, but his request goes to a database replica that is lagging, and so his phone shows that the game is still ongoing.

If Alice and Bob had hit *reload* at the same time, it would have been less surprising if they had gotten two different query results, because they wouldn't know at exactly







other requests, it would be okay for B's read to return 2. However, client A has already read the new value 4 before B's read started, so B is not allowed to read



iii. Strictly speaking, ZooKeeper and etcd provide linearizable writes, but reads may be stale, since by default they can be served by any one of the replicas. You can optionally request a linearizable read: etcd calls this a *quorum read* [16], and in ZooKeeper you need to call `sync()` before the read [15]; see “[Implementing linear](#)

This situation is strikingly similar to a lock: when a user registers for your service, you can think of them acquiring a “lock” on their chosen username. The operation is also very similar to an atomic compare-and-set, setting the username to the ID of the user who claimed it, provided that the username is not already taken.

Similar issues arise if you want to ensure that a bank account balance never goes negative, or that you don’t sell more items than you have in stock in the warehouse, or that two people don’t concurrently book the same seat on a flight or in a theater. These constraints all require there to be a single up-to-date value (the account balance, the stock level, the seat occupancy) that all nodes agree on.

In real applications, it is sometimes acceptable to treat such constraints loosely (for example, if a flight is overbooked, you can move customers to a different flight and

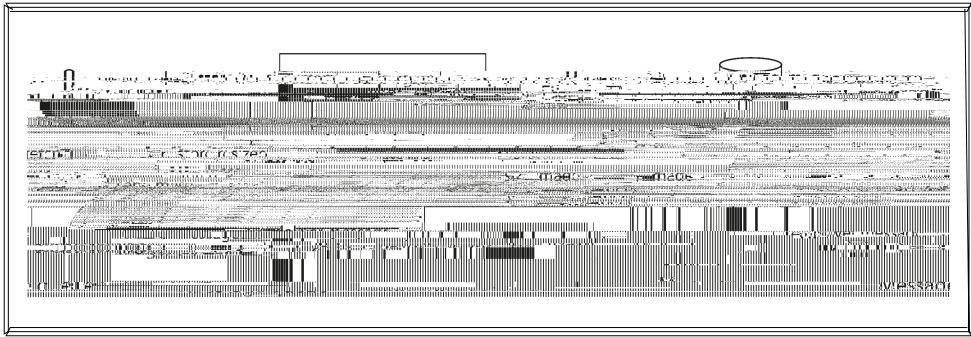


Figure 9-2. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

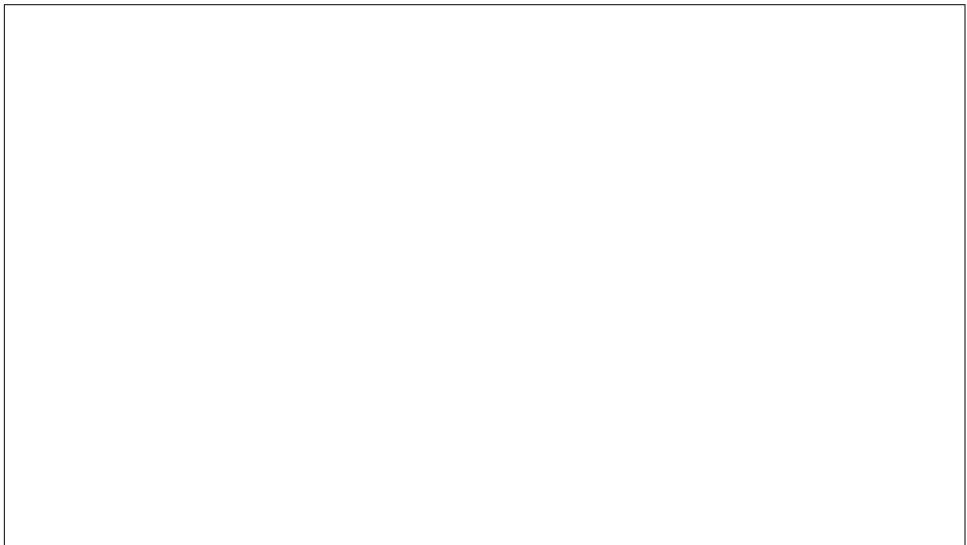
If the file storage service is linearizable, then this system should work fine. If it is not linearizable, there is the risk of a race condition: the message queue (steps 3 and 4 in [Figure 9-2](#)) might be faster than the internal replication inside the storage service. In this case, when the resizer fetches the image (step 5), it might see an old version of the If

The most common approach to making a system fault-tolerant is to use replication. Let's revisit the replication methods from [Chapter 5](#), and compare whether they can be made linearizable:

Single-leader replication (potentially linearizable)

In a system with single-leader replication (see “[Leaders and Followers](#)” on page [152](#)), the leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes. If you make readsTw 17o174513 T

ter (266 „[Distributed Transactions and Consensus](#)“ on page 323).
Since it is only a single-object consistency guarantee. Cross-primary transactions are a different matter.
It's important to note that there is a separate layer below replication, does not





Ordering Guarantees

We said previously that a linearizable register behaves as if there is only a single copy of the data, and that every operation appears to take effect atomically at one point in time. This definition implies that operations are executed in some well-defined order. We illustrated the ordering in [Figure 9-4](#) by joining up the operations in the order in which they seem to have executed.

Ordering has been a recurring theme in this book, which suggests that it might be an important fundamental idea. Let's briefly recap some of the other contexts in which we have discussed ordering:

- In [Chapter 5](#) we saw that the main purpose of the leader in single-leader replication is to determine the *order of writes* in the replication log—that is, the order in which followers apply those writes. If there is no single leader, conflicts can occur due to concurrent operations (see “[Handling Write Conflicts](#)” on page 171).

If a system obeys the ordering imposed by causality, we say that it is *causally consistent*. For example, snapshot isolation provides causal consistency: when you read from the database, and you see some piece of data, then you must also be able to see any data that causally precedes it (assuming it has not been deleted in the meantime).

The causal order is not a total order

A *total order* allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller. For example, natural numbers are totally ordered: if I give you any two numbers, say 5 and 13, you can tell me that 13 is greater than 5.

However, mathematical sets are not totally ordered: is $\{a, b\}$ greater than $\{$

In order to maintain causality, you need to know which operation

Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a *total order*: that is, every operation has a unique sequence number, and you can always compare two sequence numbers to determine which is greater (i.e., which operation happened later).

In particular, we can create sequence numbers in a total order that is *consistent with causality*:^{vii} we promise that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B). Concurrent operations may happen arbitrarily. Such a total order captures all the causality on page

rations that is consistent

reorder 1.343849775 Tw f782299 -12.6000tlicatioTd(), tif wrfd(), land Fol Thiscan alwn A caus0 Td(consis(isn thais th

vii. A total order that is *inconsistent* with causality is easy to create, but not very useful. For example, you can generate a random UUID for each operation, and compare UUIDs lexicographically to define the total order of operations. This is a valid total order, but the random UUIDs tell you nothing about which operation actually happened first, or whether the operations were concurrent.

course, messages will not be delivered while the network is interrupted, but an algo

with Lamport timestamps—in fact, this is the key difference between total order broadcast and timestamp ordering.

How hard could it be to make a linearizable integer with an atomic increment-and-get operation? As usual, if things never failed, it would be easy: you could just keep it in a variable on one node. The problem lies in handling the situation when network connections to that node are interrupted, and restoring the value when that node fails [59]. In general, if you think hard enough about linearizable sequence number generators, you inevitably end up with a consensus algorithm.

This is no coincidence: it can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both *equivalent to consensus* [28, 67]. That is, if you can solve one of these problems, you can transform it into a solution for the others. This is quite a profound and surprising insight!

It is time to finally tackle the consensus problem head-on, which we will do in the rest of this chapter.

Distributed Transactions and Consensus

Consensus is one of the most important and fundamental problems in distributed computing. On the surface, it seems simple: informally, the goal is simply to *get several nodes to agree on something*. You might think that this shouldn't be too hard. Unfortunately, many broken systems have been built in the mistaken belief that this problem is easy to solve.

Although consensus is very important, the section about it appears late in this book because the topic is quite subtle, and appreciating the subtleties requires some prerequisite knowledge. Even in the academic research community, the understanding of consensus only gradually crystallized over the course of decades, with many misunderstandings along the way. Now -we have discussed replication (Chapter 5), transactions (Chapter 7), system models (Chapter 8), linearizability, and total order broadcast (this chapter), we are finally ready to tackle the consensus problem.

There are a number of situations in which it is important for nodes to agree. For example:

Leader election



Atomic Commit and Two-Phase Commit (2PC)

In



happen that the commit succeeds on some nodes and fails on other nodes, which would violate the atomicity guarantee:

-



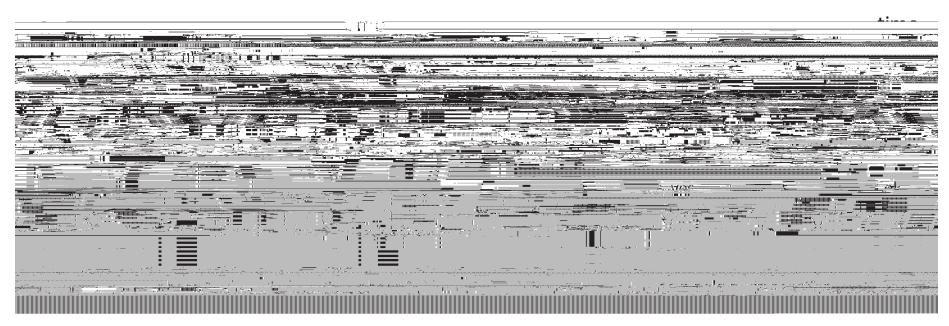


Figure 9-9. A successful execution of two-phase commit (2PC).



Don't confuse 2PC and 2PL

Two-phase *commit* (2PC) and two-phase *locking* (see “[Two-Phase Locking \(2PL\)](#)” on page 257) are two very different things. 2PC provides atomic commit in a distributed database, whereas 2PL provides serializable isolation. To avoid confusion, it’s best to think of them as entirely separate concepts and to ignore the unfortunate similarity in the names.

2PC uses a new component that doesn’t normally appear in single-node transactions: a *coordinator* (also known as *transaction manager*). The coordinator is often

acknowledgments, the ministra8s 1 72 61gouncesminiscouple husband and wife:ministrans

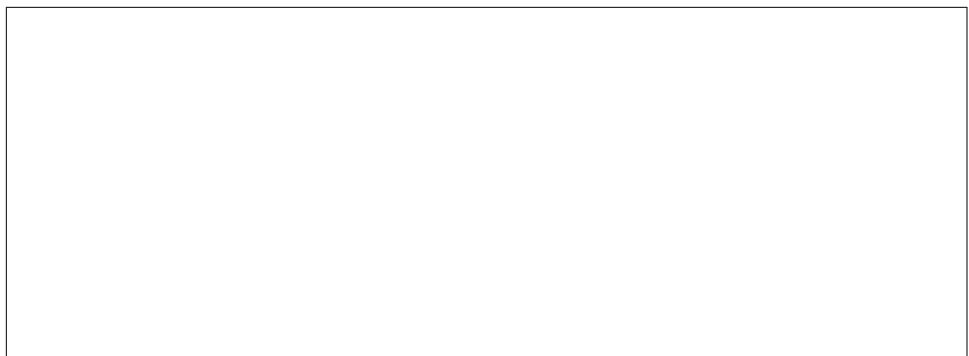


Figure 9-10. The coordinator crashes after participam6 72 dr2 1“yes.Do atabase 0 d 1. Tot



Recovering from coordinator failure

In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions. However, in practice, *orphaned* in-doubt transactions do occur [89, 90]—that is, transactions for which the coordinator cannot decide the outcome for whatever reason (e.g., because the transaction log has been lost or corrupted due to a software bug). These transactions cannot be resolved automatically, so they sit forever in the database, holding locks and blocking

In this formalism, a consensus algorithm must satisfy the following properties [25]:^{xiii}

Uniform agreement

xiii. This particular variant of consensus is called *uniform consensus*, which is equivalent to regular consensus in asynchronous systems with unreliable failure detectors [71]. The academic literature usually refers to *processes* rather than *nodes*, but we use *nodes* here for consistency with the rest of this book.

Of course, if *all* nodes crash and none of them are running, then it is not possible for any algorithm to decide anything. There is a limit to the number of failures that an algorithm can tolerate: in fact, it can be proved that any consensus algorithm requires

Epoch numbering and quorums

All of the consensus protocols discussed so far internally use a leader in some form or another, but they don't guarantee that the leader is unique. Instead, they can make a weaker guarantee: the protocols define an *epoch number* (called the *ballot number* in Paxos, *view number* in Viewstamped Replication, and *term number* in Raft) and guarantee that within each epoch, the leader is unique.

Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus epoch numbers are totally ordered and monotonically increasing. If there is a conflict between two different leaders in two different epochs (perhaps because the previous leader actually wasn't dead after all), then the leader with the higher epoch number prevails.

Before a leader is allowed to decide anything, it must first check that there isn't some other leader with a higher epoch number which might take a conflicting decision. How does a leader know that it hasn't been ousted by another node? Recall "[The Truth Is Defined by the Majority](#)" on page 300: a node cannot necessarily trust its own judgment—just because a node thinks that it is the leader, that does not necessarily mean the other nodes accept it as their leader.

Instead, it must collect votes from a *quorum* of nodes (see "[Quorums for reading and writing](#)" on page 179). For every decision that a leader wants to make, it must send off nodee a response to a leader with a higher epoch.

leader with a higher epoch.

Limitations of consensus

Consensus algorithms are a huge breakthrough for distributed systems: they bring



Failure detection

Clients maintain a long-lived session on ZooKeeper servers, and the client and server periodically exchange heartbeats to check that the other node is still alive. Even if the connection is temporarily interrupted, or a ZooKeeper node fails, the session remains active. However, if the heartbeats cease for a duration that is longer than the session timeout, ZooKeeper declares the session to be dead. Any locks held by the client are automatically released when the session times out (ZooKeeper calls these *ephemeral nodes*).

Change notifications

Not only does a client read locks and values that were created by another client, but it also watches them for changes. Thus, a client finds out when another client joins the cluster (based on the value it writes to ZooKeeper), or if another client fails (because its session times out and its ephemeral nodes disappear). By subscribing to notifications, a client avoids having to frequently poll to find out about changes.

Of these features, only the linearizable atomic operations really require consensus. However, it is the combination of these features that makes systems like ZooKeeper so useful for distributed coordination.

Allocating work to nodes

One example in which the ZooKeeper/Chubby model works well is if you have several

An application may initially run only on a single node, but eventually may grow to thousands of nodes. Trying to perform majority votes over so many nodes would be terribly inefficient. Instead, ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients. Thus, ZooKeeper provides a way of “outsourcing” some of the work of coordinating nodes (consensus, operation ordering, and failure detection) to an external service.

Normally, the kind of data managed by ZooKeeper is quite slow-changing: it represents information like “the node running on 10.1.1.23 is the leader for partition 7.”



A membership service determines which nodes are currently active and live members of a cluster. As we saw throughout [Chapter 8](#), due to unbounded network delays it's not possible to reliably detect whether another node has failed. However, if you couple failure detection with consensus, nodes can come to an agreement about which nodes should be considered alive or not.

It could still happen that a node is incorrectly declared dead by nsus, nodes even though it is actually alive. But it is nevertheless very useful for a system to have agreement on which nodes sus,titute the current membership. For example, choosing a leader could mean simply choosing the lowest-numbered among the current members, but this approach would not work if different nodes have divergent opinions on who the current members are.

Summary

In this chapter we examined the topics of consistency and sus, node from several dif

Linearizable compare-and-set registers

The register needs to atomically *decide* whether to set its value, based on whether

3. Use an algorithm to automatically choose a new leader. This approach requires a consensus algorithm, and it is advisable to use a proven algorithm that correctly handles adverse network conditions [107].

Although a single-leader database can provide linearizability without executing a consensus algorithm on every write, it still requires consensus to maintain its leadership and for leadership changes. Thus, in some sense, having a leader only “kicks the can down the road”: consensus is still required, only in a different place, and less frequently. The good news is that fault-tolerant algorithms and systems for consensus exist, and we briefly discussed them in this chapter.

Tools like ZooKeeper play an important role in providing an “outsourced” consensus, failure detection, and membership service that applications can use. It’s not easy to use, but it is much better than trying to develop your own algorithms that can withstand all the problems discussed in [Chapter 8](#). If you find yourself wanting to do one of those things that is reducible to consensus, and you want it to be fault-tolerant, then it is advisable to use something like ZooKeeper.

Nevertheless, not every system necessarily requires consensus: for example, leaderless and multi-leader replication systems typically do not use global consensus. The conflicts that occur in these systems (see [“Handling Write Conflicts” on page 171](#)) are a consequence of not having consensus across different leaders, but maybe that’s okay: maybe we simply need to cope without linearizability and learn to work better with data that has branching and merging version histories.

This chapter referenced a large body of research on the theory of distributed systems. Although the theoretical papers and proofs are not always easy to understand, and sometimes make unrealistic assumptions, they are incredibly valuable for informing practical work in this field: they help us reason about what can and cannot be done, and help us find the counterintuitive ways in which distributed systems are often flawed. If you have the time, the references are well worth exploring.

This brings us to the end of [Part II](#) of this book, in which we covered replication ([Chapter 5](#)), part 5.12 ing ([Chapter 6](#)), transactions ([Chapter 7](#)), distributed system failure models ([Chapter 8](#)), and finally consistency and consensus ([Chapter 9](#)). Now that we have laid a firm foundation of theory, in [Part III](#) we will turn once again to more practical systems, and discuss how to build powerful applications from hetero

- [2] Prince Mahajan, Lorenzo Alvisi, and Mijo5hahlin: “**Consistency, Availability, and Convergence**,” University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011.
- [3] Alex Scotti: “**Adventures in Building Your Own5hatabase**,” at *All Your Base*, November 2015.
- [4] Peter Bailis, Aaron5havidson, Alan Fekete, et al.: “**Highly Available Transactions: Virtues and Limitations**,” at *40th International Conference on5Very Large5hata Bases* (VLDB), September 2014. Extended version5publishmbeas pre-print arXiv:1302.0309 [cs.DB].
- [5] Paolo Viotti and Marko Vukoli : “**Consistency in Non-Transactional Distributed Storage5Systems**” arXiv:1512.00168, 12 April 2016.
- [6] Maurice P. Herlihy and Jeannette M. Wing: “**Linearizability: A Correctness Condition for Concurrent Objects**,” *ACM Transactions on5Programming Languages and Systems* (TOPLAS), volume 12, number 3, pages 463–492, July 1990. doi: 10.1145/78969.78972
- [7] Leslie Lamport: “**On, in Distributed Computing**” in *Distributed Computing*, volume 2, pages 77–101, June 1986. doi:10.1007/BF01786228

Gifford: “**Information Storage5in a Decentralized Computer System**,”

[

- [17] “Apache Curator,” Apache Software Foundation, *curator.apache.org*, 2015.
- [18] Morali Vallath: *Oracle 10g RAC Grid, Services & Clustering*. Elsevier Digital Press, 2006. ISBN: 978-1-555-58321-7
- [19] Peter Bailis, Alan Fekete, Michael J Franklin, et al.: “**Coordination-Avoiding Database Systems**,” *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.
- [20] Kyle Kingsbury: “**Call Me Maybe: etcd and Consul**,” *aphyr.com*, June 9, 2014.
- [21] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini: “**Zab: High-Performance Broadcast for Primary-Backup Systems**,” at *41st IEEE International*



[44] Martin Thompson: “Memory Barriers/Fences,” *mechanical-sympathy.blogspot.co.uk*, July 24, 2011.

[45] Ulrich Drepper: “What Every Programmer Should Know About Memory,” *akkadia.org*, November 21, 2007.

[46] Daniel J. Abadi: “Consistency Tradeoffs in Modern Distributed Database System Design,” *IEEE Computer Magazine*, volume 45, number 2, pages 37–42, February 2012. doi:[10.1109/MC.2012.33](https://doi.org/10.1109/MC.2012.33)

[47] Hagit Attiya and Jennifer L. Welch: “Sequential Consistency Versus Linearizability,” *ACM Transactions on Computer Systems* (TOCS), volume 12, number 2, pages 91–122, May 1994. doi:[10.1145/176575.176576](https://doi.org/10.1145/176575.176576)

[

[57] Xavier Défago, André Schiper, and Péter Urbán: “**Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey**,” *ACM Computing Surveys*, volume 36, number 4, pages 372–421, December 2004. doi:[10.1145/1041680.1041682](https://doi.org/10.1145/1041680.1041682)

[58] Hagit Attiya and Jennifer Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edition. John Wiley & Sons, 2004. ISBN: 978-0-471-45324-6, doi:[10.1002/0471478210](https://doi.org/10.1002/0471478210)

[59] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, et al.: “**CORFU: A**



[82] Pat Helland: “[Life Beyond Distributed Transactions: An Apostate’s Opinion](#),” at

[97] Leslie Lamport: “**Paxos Is a Simple** ,” *ACM SIGACT News*, volume 32, number 4, pages 51–58, December 2001.

[98] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: “



Systems of Record and Derived Data

On a high level, systems that store and process data can be grouped into two broad categories:

Systems of record

A system of record, also known as *source of truth*, holds the authoritative version

CHAPTER 10

Batch Processing

A system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fair On robust, the real test begins as people with many different viewpoints undertake their own experiments.

—Donald Knuth

In the first two parts of this book we talked a lot about *requests* and *queries*, and the corresponding *responses* or *results*. This style of data processing is assumed in many modern data systems: you ask for something, or you send an instruction, and some time later the system (hopefully) gives you an answer. Databases, caches, searchtir(modern work

while because the ideas and lessons from Unix carry over to large-scale, heterogeneous distributed data systems.

i. Some people love to point out that cat is unnecessary¹. However, it is a fine example given moderately and unwritten like this.

same principle as we discussed in “[SSTables and LSM-Trees](#)” on page 76: chunks of data can be sorted in memory and written out to disk as segment files, and then multiple sorted segments can be merged into a larger sorted file. Mergesort has sequential

The parsing of each record (i.e., a line of input) is more vague. Unix tools commonly split a line into fields by whitespace or tab characters, but CSV (comma-separated), pipe-separated, and other encodings are also used. Even a fairly simple tool like `xargs` has half a dozen command-line options for specifying how its input should be parsed.

The uniform interface of ASCII text mostly works, but it's not exactly beautiful: our log analysis example used `{print $7}` to extract the URL, which is not very readable. In an ideal world this could have perhaps been `{print $request_url}` or something of that sort. We will return to this idea later.

Although it's not perfect, even decades later, the uniform interface of Unix is still something remarkable. Not many pieces of software interoperate and compose as well as Unix tools do: you can't easily pipe the contents of your email account and your online shopping history through a custom analysis tool into a spreadsheet and post the results to a social network or a wiki. Today it's an exception, not the norm, to have programs that work together as smoothly as Unix tools do.

Even databases with the



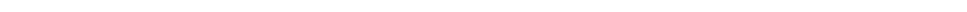
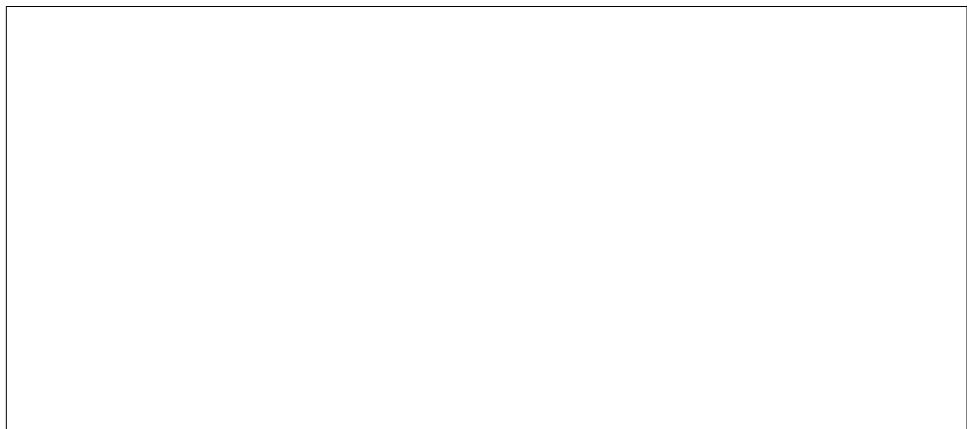
iv. One difference is that with HDFS, computing tasks can be scheduled to run on the machine that stores a copy of a particular file, whereas object stores usually keep storage and computation separate. Reading from a local disk has a performance advantage if network bandwidth is a bottleneck. Note however that if erasure coding is used, the locality advantage is lost, because the data from several machines must be combined in order to reconstitute the original file [20].

Reducer

The MapReduce framework takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an iterator over that collection of values. The reducer can produce output records (such as the number of occurrences of the same URL).

In the web server log example, we had a second sort command in step 5, which ranked URLs by number of requests. In MapReduce, if you need a second sorting stage, you can implement it by writing a second MapReduce job and using the output of the first job as input to the second job. Viewed like this, the role of the mapper is to

When we talk about joins in the context of batch processing, we mean resolving all



records with the same key form a group, and the next step is often to perform some kind of aggregation within each group—for example:

- Counting the number of records in each group (like in our example of counting page views, which you would express as a `COUNT(*)` aggregation in SQL)
- Adding up the values in one particular field (`SUM(fieldname)`) in SQL
- Picking the top k



This approach only works if both of the join's inputs have the same number of partitions, with records assigned to partitions based on the same key and the same hash

- If you introduce a bug into the code and the output is wrong or corrupted, you can simply roll back to a previous version of the code and rerun the job, and the output will be correct again. Or, even simpler, you can keep the old output in a different directory and simply switch back to it. Databases with read-write transactions do not have this property -12.6, you intend to deploy ingyde and thaille track d dbasj 0.76752478 -



(which happens to always run the `sort` utility between the map phase and the reduce phase). We saw how you can implement various join and grouping operations on top of these primitives.

When the MapReduce paper [1] was published, it was—in some sense—not at all



Beyond MapReduce

Although MapReduce became very popular and received a lot of hype in the late



Dataflow engines

In order to fix these problems with MapReduce, several new execution engines for distributed batch computations were developed, the most well known of which are Spark [61, 62], Tez [63, 64], and Flink [65, 66]. There are various differences in the way they are designed, but they have one thing in common: they handle an entire workflow as one job, rather than breaking it up into independent subjobs.

exchanged through a shared memory buffer rather than having to copy it over the network.

- It is usually sufficient for intermediate state between operators to be kept in



Graphs and Iterative Processing

In “[Graph-Like Data Models](#)” on page 49 we discussed using graphs for modeling data, and using graph query languages to traverse the edges and vertices in a graph. The discussion in [Chapter 2](#) was focused around OLTP-style use: quickly executing queries to find a small number of vertices matching certain criteria.

It is also interesting to look at graphs in a batch processing context, where the goal is



This fault tolerance is achieved by periodically checkpointing the state of all vertices

As discussed previously, higher-level languages and APIs such as Hive, Pig, Cascad



Also useful are spatial algorithms such as *k-nearest neighbors*

this case, a job is never complete, because at any time there may still be more work coming in. We shall see that stream and batch processing are similar in some respects, but the assumption of unbounded streams also changes a lot about how we build systems.

References

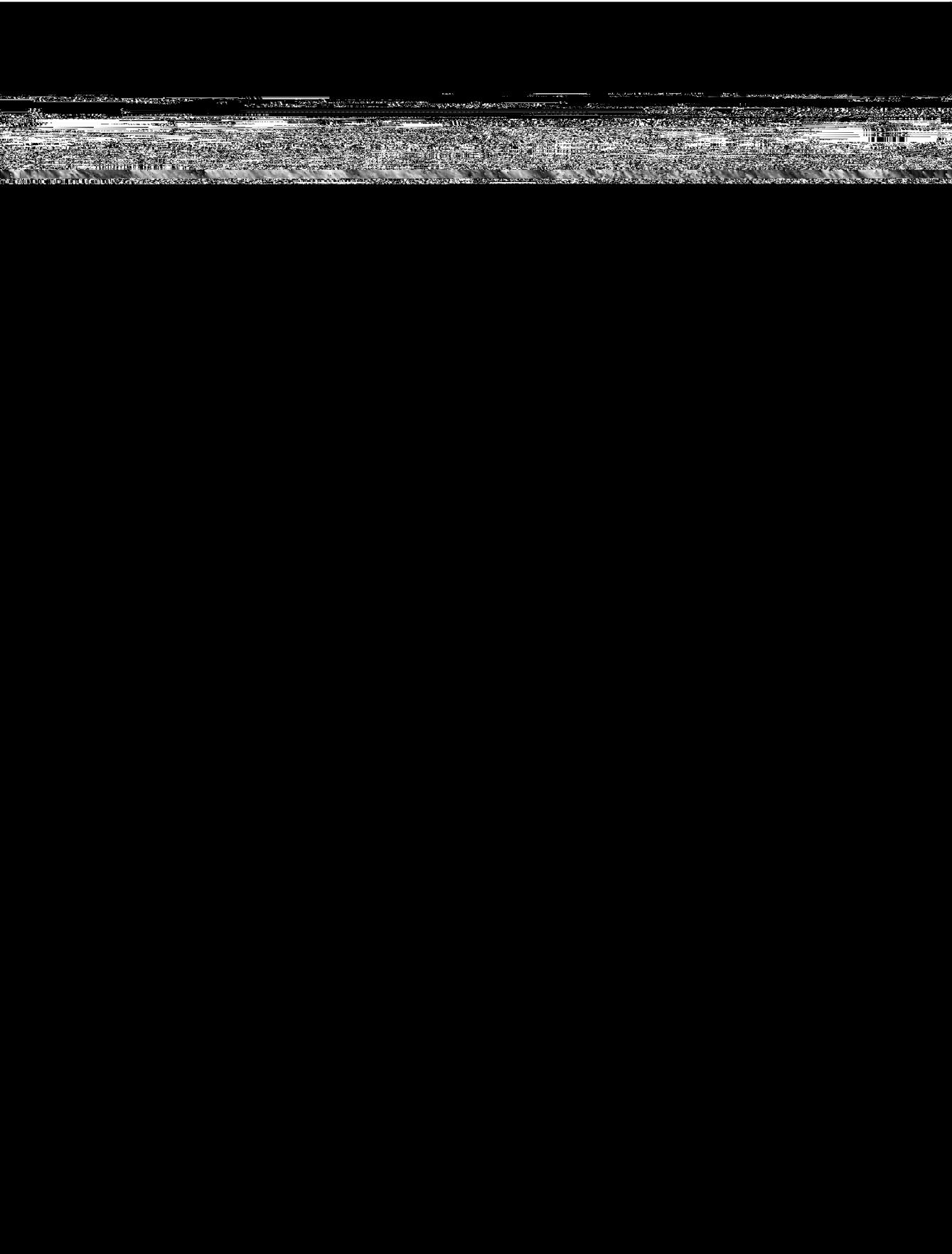
- [1] Jeffrey Dean and Sanjay Ghemawat: “[MapReduce: Simplified Data Processing on Large Clusters](#)” *USENIX Symposium on Operating System Design and Implementation*, December 2004.

Spolsky: “[The Devil’s Advocate](#)” *Joel on Software*

- [33] “Apache Crunch User Guide,” Apache Software Foundation, crunch.apache.org.
- [34] Craig Chambers, Ashish Raniwala, Frances Perry, et al.: “[FlumeJava: Easy, Efficient Data-Parallel Pipelines](#),” at *31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2010. doi: [10.1145/1806596.1806638](https://doi.org/10.1145/1806596.1806638)
- [35] Jay Kreps: “[Why Local State is a Fundamental Primitive in Stream Processing](#),” oreilly.com, July 31, 2014.
- [36] Martin Kleppmann: “[Rethinking Caching in Web Apps](#),” martin.kleppmann.com, Octoran 1, 2012.
- [37] Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira: *Hadoop Application Architectures*. O'Reilly Media, 2015. ISBN: 978-1-491-92511-0. Td700 046reOa7B-1 0 8.6835 T

[48] Nathan Marz: “[ElephantDB](#),” *slideshare.net*, May 30, 2011.

[49]



CHAPTER 11

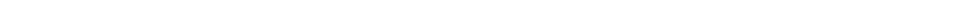
Stream Processing

slices entirely and simply processing every event as it happens. That is the idea behind *stream processing*

it fills up, the sender is blocked until the recipient takes data out of the buffer (see “[Network congestion and queueing](#)” on page 282).

If messages are buffered in a queue, it is important to understand what happens as that queue grows. Does the system crash if the queue no longer fits in memory, or does it write messages to disk? If so, how does the disk access affect the performance of the messaging system [6]?

2. *What happens if nodes crash or temporarily go offline—are any messages lost?*



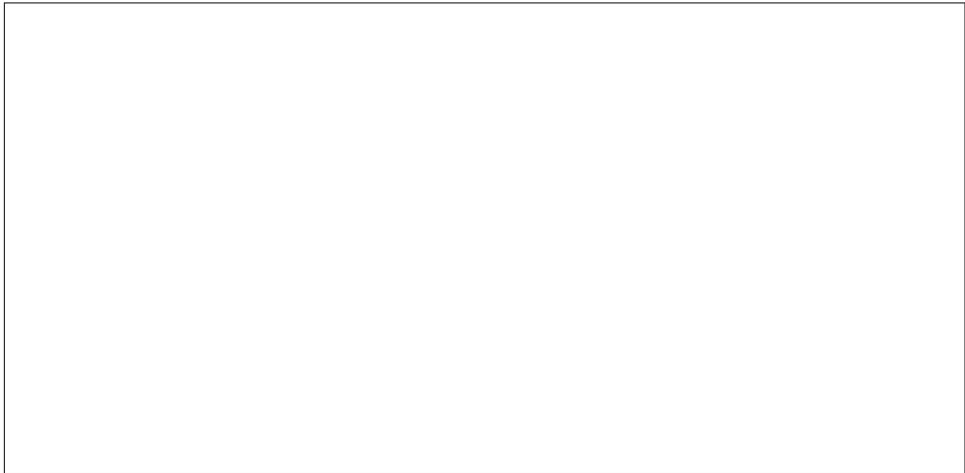
- If the consumer exposes a service on the network, producers can make a direct



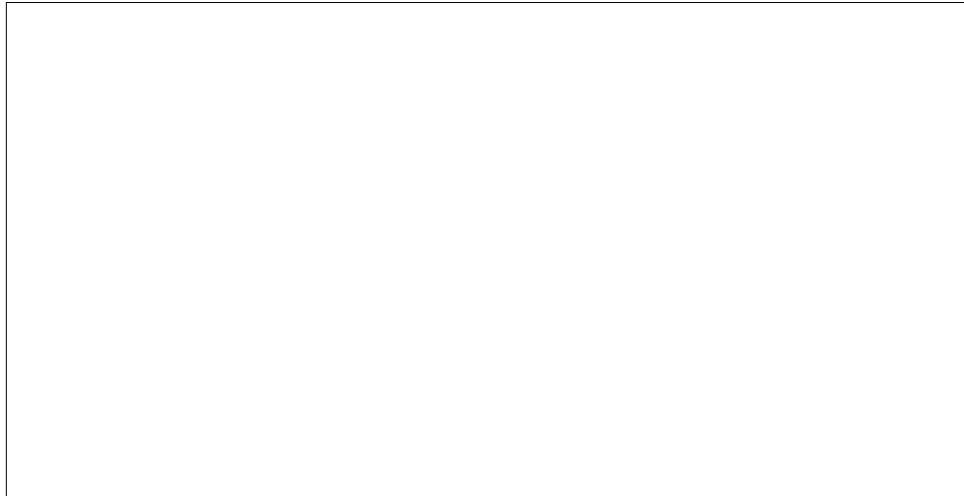
them quite similar in nature to databases, although there are still important practical differences between message brokers and databases:

- Databases usually keep data until it is explicitly deleted, whereas most message brokers automatically delete a message when it has been successfully delivered to its consumers. Such message brokers are not suitable for long-term data storage.
- Since they quickly delete messages, most message brokers assume that their

Fan-out



When combined with load balancing, this redelivery behavior has an interesting effect on the ordering of messages. In



Databases and filesystems take the opposite approach: everything that is written to a database or file is normally expected to be permanently recorded, at least until some one explicitly chooses to delete it again.



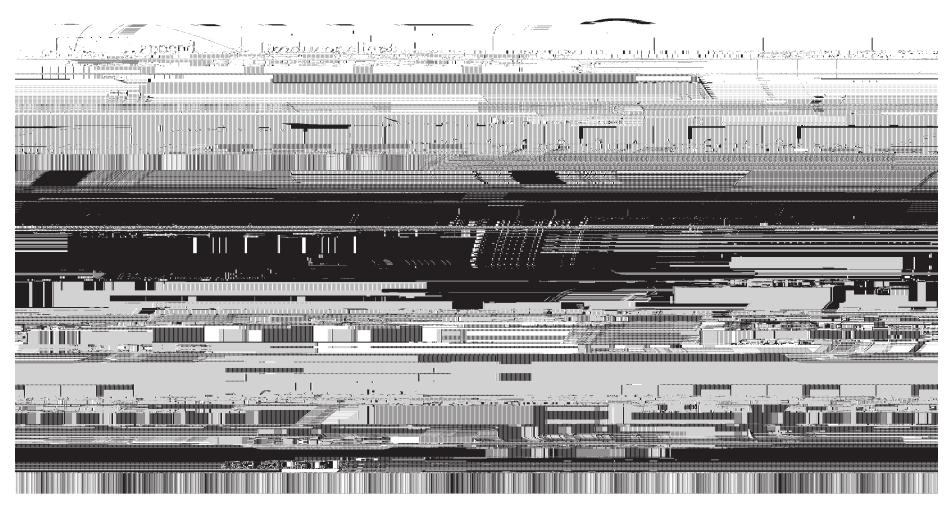


Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

Apache Kafka [17, 18], Amazon Kinesis Streams [19], and Twitter’s DistributedLog [20, 21] are log-based message brokers that work like this. Google Cloud Pub/Sub is architecturally similar but exposes a JMS-style API rather than a log abstraction [16]. Even though these message brokers write all messages to disk, they are able to achieve throughput of millions of messages per second by partitioning across multiple machines, and fault tolerance by replicating messages [22, 23].

Logs compared to traditional messaging

The log-based approach trivially supports fan-out messaging, because several consumers can independently read the log without affecting each other—reading a mes

-
- i. It's possible to create a load balancing scheme in which two consumers share the work of processing a partition by having both read the full set of messages, but one of them only considers messages with even-numbered offsets while the other deals with the odd-numbered offsets. Alternatively, you could spread message processing over a thread pool, but that approach complicates consumer offset management. In general,
-

Disk space usage

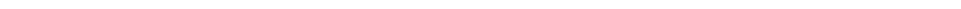
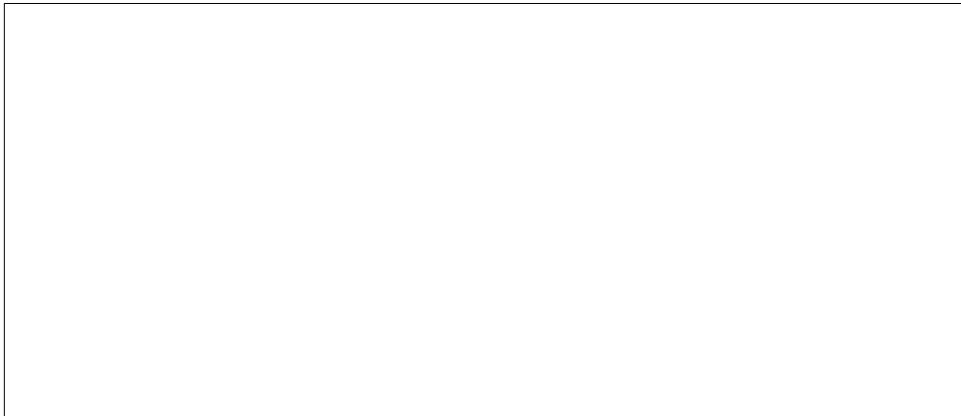
Even if a consumer does fall too far behind and starts missing messages, only that

Change Data Capture

The problem with most databases' replication logs is that they have long been considered to be an internal implementation detail of the database, not a public API. Clients are supposed to query the database through its data model and query language, not parse the replication logs and try to extract data from them.

For decades, many databases simply did not have a documented way of getting the log of changes written to them. For this reason it was difficult to take all the changes made in a database and replicate them (different storage technology was used).

are(made available was at remote medicalslygast they(are written.)Tj 139.704 Tw 2 -18.60001
are just



Log compaction

model as a table into which transactions can insert tuples, but which cannot be queried. The stream then consists of the log of tuples that committed transactions have written to this special table, in the order they were committed. External consumers

data is later going to be used. If a new application feature is introduced—for example, “the place is offered to the next person on the waiting list”—the event sourcing approach allows that new side effect to easily be chained off the existing event.

Event sourcing is similar to the chronicle data model [45], and there are also similarities between an event log and the fact table that you find in a star schema (see “[Stars and Snowflakes: Schemas for Analytics](#)” on page 93).

Specialized databases such as Event Store [46] have been developed to support applications using event sourcing, but in general the approach is independent of any particular tool. A conventional database or a log-based message broker can also be used to build applications in this style.

Deriving current state from the event log

An event log by itself is not very useful, because users generally expect to see the current state of a system, not the history of modifications. For example, on a shopping website, users expect to be able to see the current contents of their cart, not an append-only list of all the changes they have ever made to their cart.

Thus, applications that use event sourcing need to take the log of events (representing the data *written*

Commands and events

The event sourcing philosophy is careful to distinguish between *events* and *commands* [48]. When a request from a user first arrives, it is initially a command: at this



systems without having to modify them. Running old and new systems side by side is



action requiring data to be changed in several different places. With event sourcing, you can design an event such that it is a self-contained description of a user action.

Maintaining materialized views

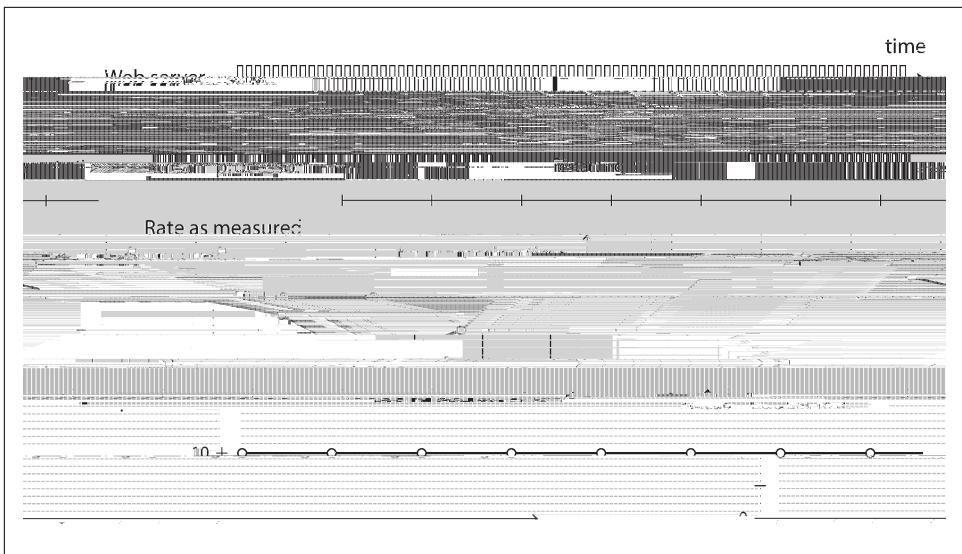
We saw in “[Databases and Streams](#)” on page 451



Message passing and RPC

In “Message-Passing Dataflow” on page 136





data pipelines to incremental processing of unbounded datasets, there is exactly the same need for joins on streams.

However, the fact that new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs. To understand the situation better, let's distinguish three different types of joins: *stream-stream* joins, *stream-table* joins, and *table-table* joins [84]. In the following sections we'll illustrate each by example.

Stream-stream join (window join)

Say you have a search feature on your website, and you want to detect recent trends in searched-for URLs. Every time someone types a search query, you log an event containing the query and the results returned. Every time someone clicks one of the search results, you log another event recording the click. In order to calculate the click-through rate for each URL in the search results, you need to bring together the events for the search action and the click action, which are connected by having the same session ID. Similar analyses are needed in advertising systems [85].

The click may never come if the user always waits for their search, and even if it comes, the time between the search and the click may be highly variable: in many cases it might be a few seconds, but it could be as long as days or weeks (if a user runs a search, forgets about that browser tab, and then returns to the tab and clicks a result some time later). Due to variable network delays, the click event may even arrive before the search event. You can choose a suitable window for the join—for example, you may choose to join a click with a search if they occur at most one hour apart.

Note that embedding the details of the search in the click event is not equivalent to joining the events: doing so would only tell you about the cases where the user clicked a search result, not about the searches where the user did not click any of the results. In order to measure search quality, you need accurate click-through rates, for which you need both the search events and the click events.

To implement this type of join, a stream processor needs to maintain *state*: for example, all the events that occurred in the last hour, indexed by session ID. Whenever a search event or click event occurs, it is added to the appropriate index, and the stream processor also checks the other index to see if another event for the same session ID has already arrived. If there is a matching event, you emit an event saying which search result was clicked. If the search event expires without you seeing a matching click event, you emit an event saying which search results were not clicked.

Stream-table join (stream enrichment)

In “[Example: analysis of user activity events](#)” on page 404 (Figure 10-2) we saw an example of a batch job joining two datasets: a set of user activity events and a database of user profiles. It is natural to think of the user activity events as a stream, and to perform the same join on a continuous basis in a stream processor: the input is a

stream of activity events containing a user ID, and the output is a stream of activity

Microbatching and checkpointing

One solution is to break the stream into small blocks, and treat each block like a miniature batch process. This approach is called *microbatching*, and it is used in Spark Streaming [91]. The batch size is typically around one second, which called the result of a performance compromise: smaller batches incur greater scheduling and coordination overhead, while larger batches mean a longer delay before results of the stream process become visible.

Microbatching also implicitly provides a tumbling window equal to the batch size (windowed by processing time, not event timestamps); any job that requires large windows need to explicitly carry over state from one microbatch to the next.

A variant approach, used in Apache Flink, is to periodically generate rolling checkpoints of state and write them to durable storage [92, 93]. If a stream operator crashes, it can restart from its most recent checkpoint and discard any output generated between the last checkpoint and the crash. The checkpoints are triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size.

Within the confines of the stream processing framework, microbatching and checkpointing approaches provide the same exactly-once semantics as batch processing. However, as soon as output leaves the stream processor (for example, by writing to a database, sending messages to an external message broker, or sending emails), the framework is no longer able to discard the output of a failed batch. In this case, restarting a failed task causes the external side effect to happen twice, and micro-

two join inputs may in fact be the same stream (a *self-join*) if you want to find related events within that one stream.

Stream-table joins

One input stream consists of activity events, while the other is a database change log. The changelog keeps a local copy of the database up to date. For each activity event, the join operator queries the database and outputs an enriched activity event.

Table-table joins

Both input streams are database changelogs. In this case, every change on one side is joined with the latest state of the other side. The result is a stream of changes to the materialized view of the join between the two tables.

[7] Vicent Martí: “

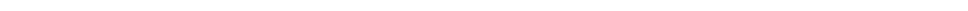
- [25] Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: “**All Aboard the Data bus!**,” at *3rd ACM Symposium on Cloud Computing* (SoCC), October 2012.
- [26] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: “**Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services**,” at



[44] Vaughn Vernon: *Implementing Domain-Driven Design*. Addison-Wesley Profes

[78] “Apache Storm 1.0.1 Documentation,” *storm.apache.org*, May 2016.

[79] Tyler Akidau: “The World Beyond Batch: Streaming 102,” *oreilly.com*, January



[95] Paris Carbone, Gyula Fóra, Stephan Ewen, et al.: “**Lightweight Asynchronous Snapshots for Distributed Dataflows**,” arXiv:1506.08603 [cs.DC], June 29, 2015.

[96] Ryan Betts and John Hugg: *Fast Data: Smart and at Scale*. Report, O'Reilly Media, October 2015.

[95] Flavio Junqueira: “**Making Sense of Exactly-Once Semantics**,” at *Strata+Hadoop World London*, June 2016.

[96] Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram Subramanian, andson GTd(041y8h
[95] flows



CHAPTER 12

The Future of Data Systems

Data Integration

A recurring theme in this book has been that for any given problem, there are several solutions, all of which have different pros, cons, and trade-offs. For example, when discussing storage engines in



gration problem becomes harder. Besides the database and the search index, perhaps you need to keep copies of the data in analytics systems (data warehouses, or batch processing systems). What one, persistence is more obscure than point-to-point integration with two clients.



The limits of total ordering

With systems that are small enough, constructing a totally ordered event log is entirely feasible (as demonstrated by the popularity of databases with single-leader replication, which construct precisely such a log). However, as systems are scaled

partition. However, causal dependencies sometimes arise in more subtle ways (see also “[Ordering and Causality](#)” on page 339).

the appropriate outputs. Batch and stream processors are the tools for achieving this goal.

The outputs of batch and stream processes are derived datasets such as search indexes, materialized views, recommendations to show to users, aggregate metrics, and so on (see



Derived views allow *gradual*

- Since the stream pipeline and the batch pipeline produce separate outputs, they need to be merged in order to respond to user requests. This merge is fairly easy



Unbundling Databases

Viewed like this, batch and stream processors are like elaborate implementations of

querying requires mapping one data model into another, which takes some thought but is ultimately quite a manageable problem. I think that keeping the writes to sev

required for maintaining state in stream processors, and in order to serve queries for the output of batch and stream processors (see



Dataflow: Interplay between state changes and application code

Thinking about applications in terms of dataflow implies renegotiating the relationship between application code and state management. Instead of treating a database as a passive variable that is manipulated by the application, we think much more

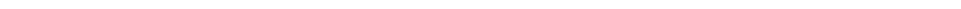


the failure of another service. The fastest and most reliable network request is no net work request at all! Instead of RPC, we now have a stream join between purchase events and exchange rate update events (see “[Stream-table join \(stream enrichment\)](#)” on page 473).

The join is time-dependent: if the purchase events are reprocessed at a later point in time, the exchange rate will have changed. If you want to reconstruct the original output, you will never obtain the historical exchange rate at the original time of purchase. No matter whether you query a service or subscribe to a stream of exchange

view, as it would need to be updated when new documents appear that should be included in the results of one of the common queries.

From this example we can see that an index is not the only possible boundary between the write path and the read path. Caching of common search results is possible, and grep-like scanning without the index is also possible on a small number of documents. Viewed like this, the role of caches, indexes, and materialized views is simple: they shift the boundary between the read path and the write path. They allow us to do more work on the write path, by precomputing results, in order to save effort on the read path.



internal client-side state by subscribing to a stream of events representing user input or responses from a server, structured similarly to event sourcing (see “[Event Sourcing](#)” on page 457).

It would be very natural to extend this programming model to also allow a server to push state-change events into this client-side event pipeline. Thus, state changes could flow through an end-to-end write path: from the interaction on one device that triggers a state change, via event logs and through several derived data systems and stream processors, all the way to the user interface of a person observing the state on another device. These state changes would be propagated with fairly low delay—say, under one second end to end.

Some applications, such as instant messaging and online games, already have such a “real-time” architecture (in the sense of interactions with low delay, not in the sense of “[Response time guarantees](#)” on page 298). But why don’t we build all applications this way?

The challenge is that the assumption of stateless clients and request/response interactions is very deeply ingrained in our databases, libraries, frameworks, and protocols. Many datastores support read/write operations where a request returns one response, but much fewer provide an ability to subscribe to changes—i.e., a request that returns a stream of responses over time (see “[API support for change streams](#)” on page 456).

In order to extend the write path all the way to the end user, we would need to fundamentally rethink the way we build many of these systems: moving away from request/response interaction and toward publish/subscribe dataflow [27]. I think that the advantages of more responsive user interfaces and better offline support would make

the nodes that store the data being queried. This is a reasonable design, but not the



Another example of this pattern occurs in fraud prevention: in order to assess the risk

behavior in the presence of network problems and crashes. Even if infrastructure

(overstating some metric). In this context, *exactly-once* means arranging the computation such that the final effect is the same as if no faults had occurred, even if the operation actually was retried due to some fault. We previously discussed a few approaches for achieving this goal.



Example 12-2 relies on a uniqueness constraint on the `request_id` column. If a



against network attackers, but not against compromises of the server. Only end-to-end encryption and authentication can protect against all of these things.

Although the low-level features (TCP duplicate suppression, Ethernet checksums,



Enforcing Constraints

Let's think about correctness in the context of the ideas around unbundling databases (“[Unbundling Databases](#)” on page 499). We saw that end-to-end duplicate suppression is a key part of ensuring correctness.

Uniqueness in log-based messaging

The log ensures that all consumers see messages in the same order—a guarantee that is formally known as *total order broadcast*



idea of using multiple differently partitioned stages is similar to what we discussed in “Multi-partition data processing” on page 514 (see also “Concurrency control” on page 462). see [“Linearizability” on page 324](#): that is, a writer waits until a transaction is committed, and thereafter its writes are immediately visible to all readers.

“Lin

If integrity is violated, the inconsistency is permanent: waiting and trying again is not going to fix database corruption in most cases. Instead, explicit checking and repair is needed. In the context of ACID transactions (see “[The Meaning of](#)

performance and operational robustness. We achieved this integrity through a combination of mechanisms:

- Representing the content of the write operation as a single message, which can easily be written atomically—an approach that fits very well with event sourcing (see “[Event Sourcing](#)” on page 457)
- Deriving all other state updates from that single message using deterministic der



erately violated for business reasons, and compensation processes (refunds, upgrades, providing a complimentary room at a neighboring hotel) are put in place to handle situations in which demand exceeds supply. Even if there was no



in some way, for example using a weak isolation level unsafely, the integrity of the database cannot be guaranteed.

Don't just blindly trust what they promise

With both hardware and software not always living up to the ideal that we would like them to be, it seems that data corruption is inevitable sooner or later. Thus, we should at least have a way of finding out if data has been corrupted so that we can fix



But then the database landscape changed: weaker consistency guarantees became the norm under the banner of NoSQL, and less mature storage technologies became widely used. Yet, because the audit mechanisms had not been developed, we continued building applications on the basis of blind trust, even though this approach had now become more dangerous. Let's think for a moment about designing for auditability.

Designing for auditability

If a transaction mutates several objects in a database, it is difficult to tell after the fact what

“Change

[Data Capture](#) on page 454), the insertions, updates, and deletions in various tables do not necessarily give a clear picture of why for -39c2 6hogies be9301 Td-194.3077 Tw -0.00304 -1inv

I could imagine integrity-checking and auditing algorithms, like those of certificate transparency and distributed ledgers, becoming more widely used in data systems in



Naturally, payment networks want to prevent fraudulent transactions, banks want to avoid bad loans, airlines want to avoid hijackings, and companies want to avoid hir

past, moral imagination is required, and that's something only humans can provide [87]. Data and models should be our tools, not our masters.



return: the relationship between the service and the user is very asymmetric and one-sided. The terms are set by the service, not by the user /99].



Even if particular users cannot be personally reidentified from the bucket of people targeted by a particular ad, they have lost their agency about the disclosure of some



mation into the surveillance infrastructure [99]. The delightful human creativity and social relationships that often find expression in online services are cynically exploited by the data extraction machine.

The assertion that personal data is a valuable asset is supported by the existence of data brokers, a shady industry operating in secrecy, purchasing, aggregating, analyzing, inferring, and reselling intrusive personal data about people, mostly for marketing purposes [90]. Startups are valued by their user numbers, by “eyeballs”—i.e., by their surveillance capabilities.

Because the data is valuable, many people want it. Of course companies want it—that’s why they collect it in the first place. But governments want to obtain it too: by

Companies that collect lots of data about people oppose regulation as being a burden and a hindrance to innovation. To some extent that opposition is justified. For exam

problem in one area is prevented from spreading to unrelated parts of the system, increasing the robustness and fault-tolerance of the system as a whole.

Expressing dataflows as transformations from one dataset to another also helps evolve applications: if you want to change one of the processing steps, for example to change the structure of an index or cache, you can just rerun the new transformation code on the whole input dataset in order to rederive the output. Similarly, if something goes wrong, you can fix the code and reprocess the data in order to recover.



[16] Dennis M. Ritchie and Ken Thompson: “**The UNIX Time-Sharing System**,” *Communications of the ACM*

[30] Evan Czaplicki and Stephen Chong: “[Asynchronous Functional Reactive Programming for GUIs](#),” at *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2013. [doi:10.1145/2491956.2462161](https://doi.org/10.1145/2491956.2462161)

[31] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn

[59] Jim Gray: “

[76] Mark D. Ryan: “[Enhanced Certificate Transparency and End-to-End Encrypted Mail](#),” at *Network and Distributed System Security Symposium* (NDSS), February 2014. doi:[10.14722/ndss.2014.23379](https://doi.org/10.14722/ndss.2014.23379)

[77] “[Software Engineering Code of Ethics and Professional Practice](#),” Association for Computing Machinery, *acm.org*



lock

A mechanism to ensure that only one thread, node, or transaction can access something, and anyone else who wants to







skew, 291-294, 334
slewing, 289
synchronization and accuracy, 289-291
synchronization using GPS, 287, 290, 294,
295

in log-based systems, 351,

concerns when designing, 5
future of, 489-544
 correctness, constraints, and integrity, 515-533
 data integration, 490-498
 unbundling databases, 499-515
heterogeneous, keeping in sync, 452
maintainability, 18-22
possible faults in, 221
reliability, 6-10
 hardware faults, 7
 human errors, 9
 importance of, 10
 software errors, 8
scalability, 10-18
unreliable clocks, 287-299
data warehousing, 91-95, 554
 comparison to data lakes, 415
 ETL (extract-transform-load), 92, 416, 452
 keeping data systems in sync, 452
 schema design, 93



Fossil (version control system), [463](#)
shunning (deleting data), [463](#)
FoundationDB (database)
 serializable transactions, [261](#), [265](#)

detecting corruption, 519, 530

faults in, 7, 227

sequential write throughputmon, 104.823 0 0 rg54.0950 Tj 04

deriving state from event log, [459-464](#)
for crash recovery, [75](#)
in B-trees, [82, 242](#)
in event sourcing, [457](#)
inputs to Unix commands, [397](#)

unreliability of, 277
ISDN (Integrated Services Digital Network),
284
isolation (in transactions), 225, 228, 555
 correctness and, 515
 for single-object writes, 230
serializability, 251-266

Large Hadron Collider (LHC),

in batch processing, 400, 405, 421
in stateful clients, 170, 511
in stream processing, 474, 478, 508, 522
location transparency, 134
in the actor model,

Meteor (web framework), 456



graph databases versus, 60
imperative query APIs, 46
Network Time Protocol (see NTP)
networks
 congestion and queueing, 282
 datacenter network topologies, 276
 faults (see faults)
 linearizability and network delays, 338
 network partitions,

partial order, 341

serializable snapshot isolation (SSI), [261](#)
snapshot isolation support, [239](#), [242](#)
WAL-based replication, [160](#)
XML and JSON support,

Rubygems (package manager), 428
rules (Datalog), 61

§
safety and liveness properties, 308
 in consensus algorithms, 366
 in transactions, 222
sagas (see compensating transactions)
Samza (stream processor), 466, 467
 fault tolerance, 479
 streaming SQL support, 466
sandboxes, 9
SAP HANA (database), 93
scalability, 10-18, 489
 approaches for coping with load, 17
 defined, 22
 describing load, 11
 describing performance, 13
 partitioning and, 199
 replication and, 161
 scaling up versus scaling out, 146
scaling out, 17, 146
 (see also shared-nothing architecture)
scaling up, 17, 146
scatter/gather approach, querying partitioned databases, 207
SCD (slowly changing dimension), 476
schema-on-read, 39
 comparison to evolvable schema, 128
 in distributed filesystems, 415
schema-on-write, 39
schemaless databases (see schema-on-read)
schemas, 557
 Avro, 122-127
 reader determining writer's schema, 125
 schema evolution, 123
 dynamically generated, 126
 evolution of, 496
 affecting application code, 111
 compatibility checking, 126
 in databases, 129-131
 in message-passing, 138
 in service calls, 136
 flexibility in document model, 39
 for analytics, 93-95
 for JSON and XML, 115
 merits of, 127
 schema migration on railways, 496
 Thrift and Protocol Buffers, 117-121

schema evolution, 120
traditional approach to design, fallacy in, 462

searches
 building search indexes in batch processes, 411
 k-nearest neighbors, 429
 on streams, 467
 partitioned secondary indexes, 206
secondaries (see leader-based replication)
secondary indexes,

serialization, [113](#)
(see also encoding)
service discovery, [135](#), [214](#), [372](#)
 using DNS, [216](#), [372](#)
service level agreements (SLAs), [15](#)
service-oriented architecture (SOA), [132](#)
 (see also services)
services, [131](#)-

implementation in ZooKeeper and etcd, 370
implementing with linearizable storage, 351
using, 349
using to implement linearizable storage, 350

tracking behavioral data, 536
(see also privacy)

transaction coordinator (see coordinator)

transaction manager (see coordinator)

transaction processing, 28, 90-95
comparison to analytics, 91
comparison to data warehousing, 93

transactions, 221-267, 558

ACID properties of, 223
atomicity, 223
consistency, 224
durability, 226
isolation, 225

compensating (see compensating transac