Slides for Chapter 21: Designing Distributed Systems: Google Case Study

Figure 21.1
Outline architecture of the original Google search engine [Brin and Page 1998]

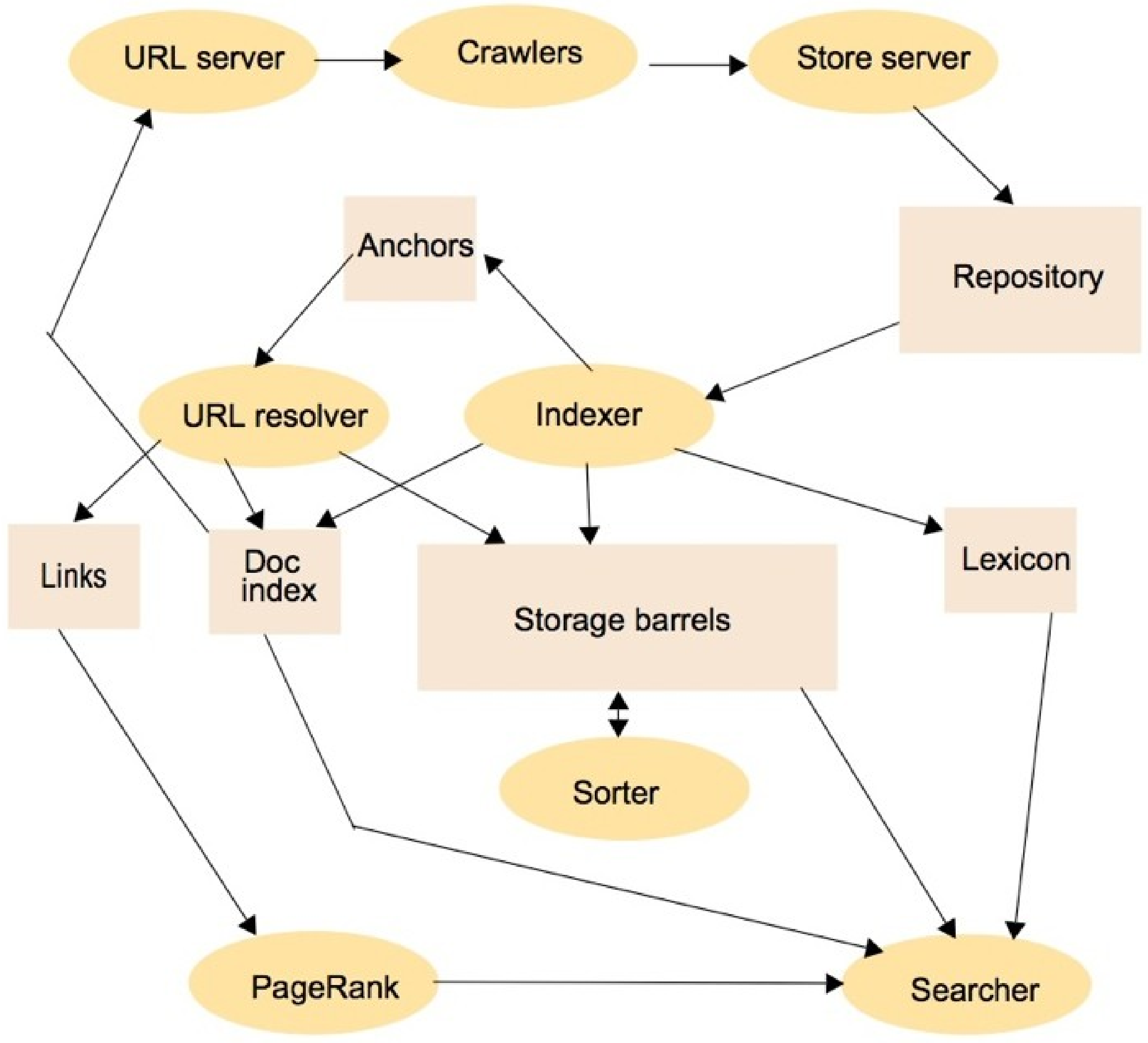
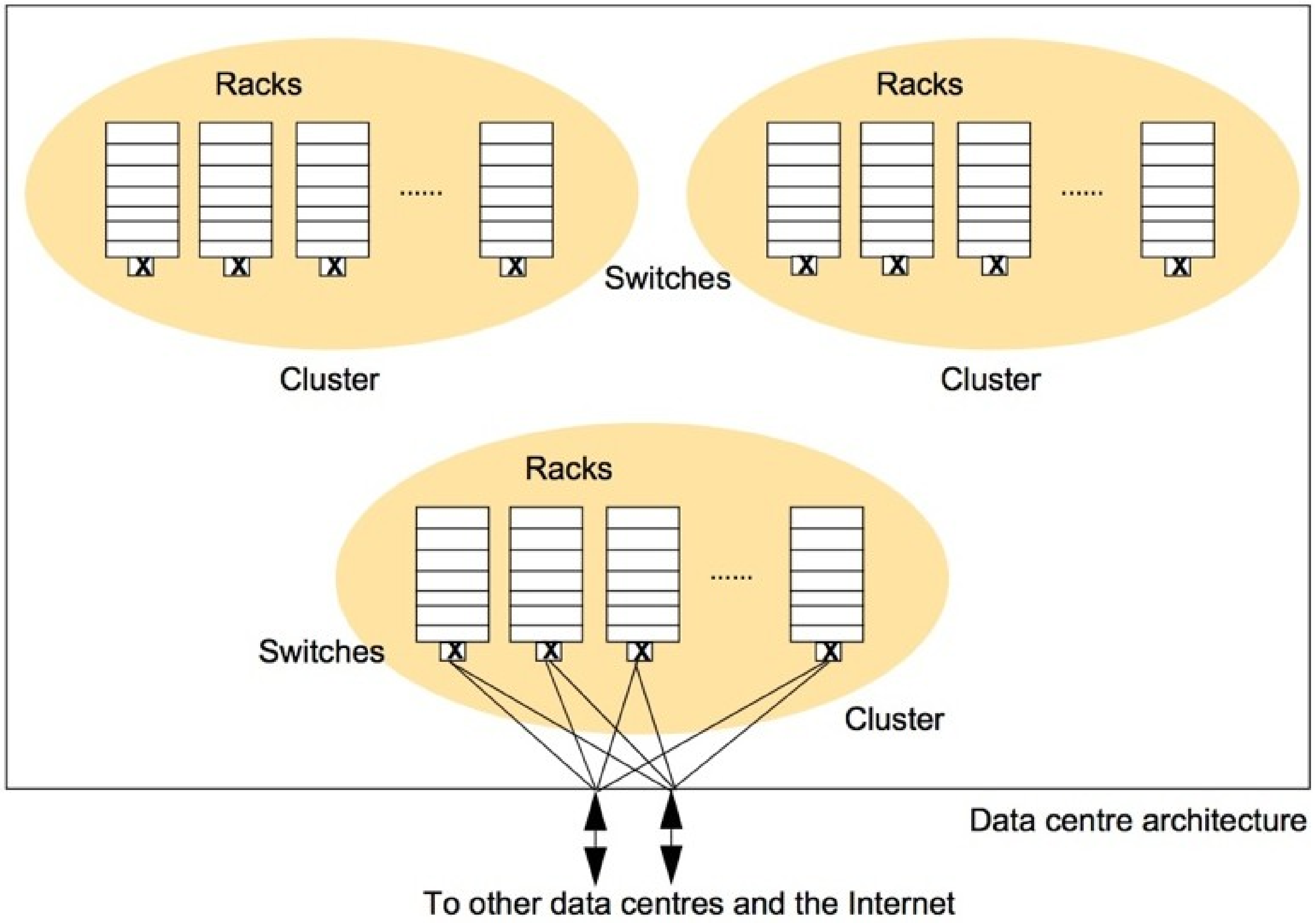


Figure 21.2
Example Google applications

Application	*Description*
Gmail	Mail system with messages hosted by Google but desktop-like message management.
Google Docs	Web-based office suite supporting shared editing of documents held on Google servers.
Google Sites	Wiki-like web sites with shared editing facilities.
Google Talk	Supports instant text messaging and Voice over IP.
Google Calendar	Web-based calendar with all data hosted on Google servers.
Google Wave	Collaboration tool integrating email, instant messaging, wikis and social networks.
Google News	Fully automated news aggregator site.
Google Maps	Scalable web-based world map including high-resolution imagery and unlimited user-generated overlays.
Google Earth	Scalable near-3D view of the globe with unlimited user-generated overlays.
Google App Engine	Google distributed infrastructure made available to outside parties as a service (platform as a service).

Figure 21.3
Organization of the Google physical infrastructure



(To avoid clutter the Ethernet connections are shown from only one of the clusters to the external links)

Figure 21.4
The scalability problem in Google

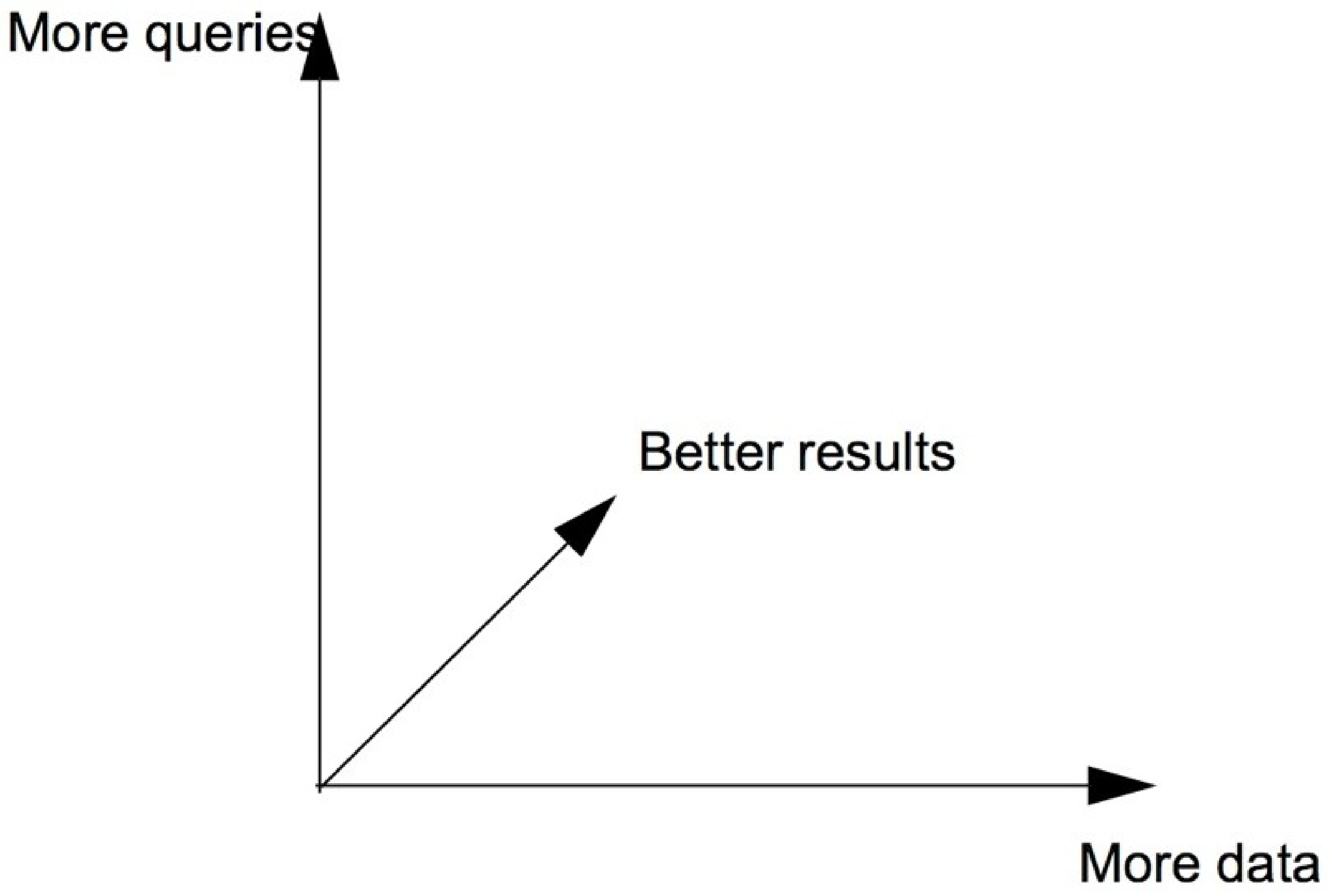


Figure 21.5
The overall Google systems architecture

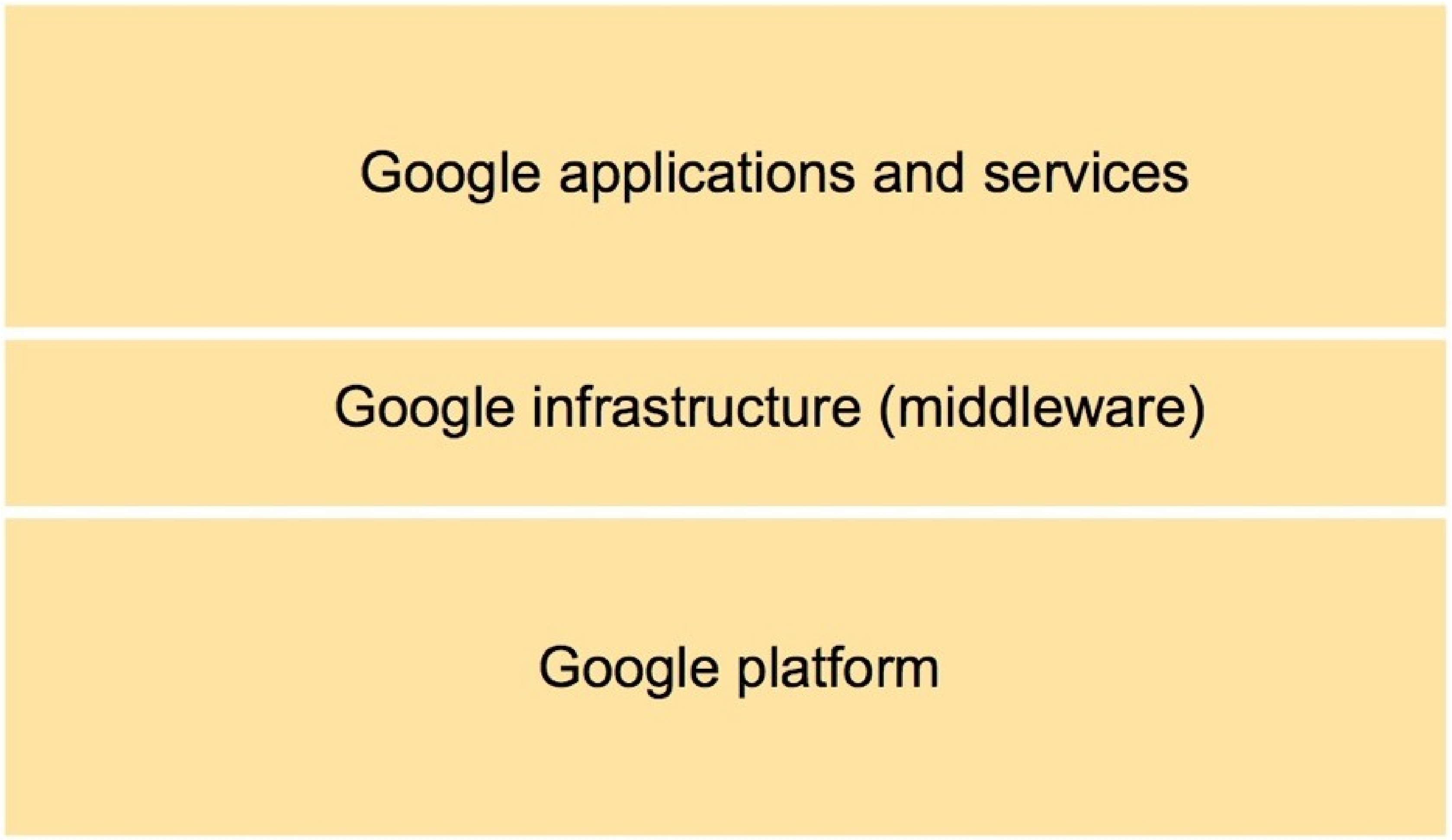

Google applications and services
Google infrastructure (middleware)
Google platform

Figure 21.6
Google infrastructure

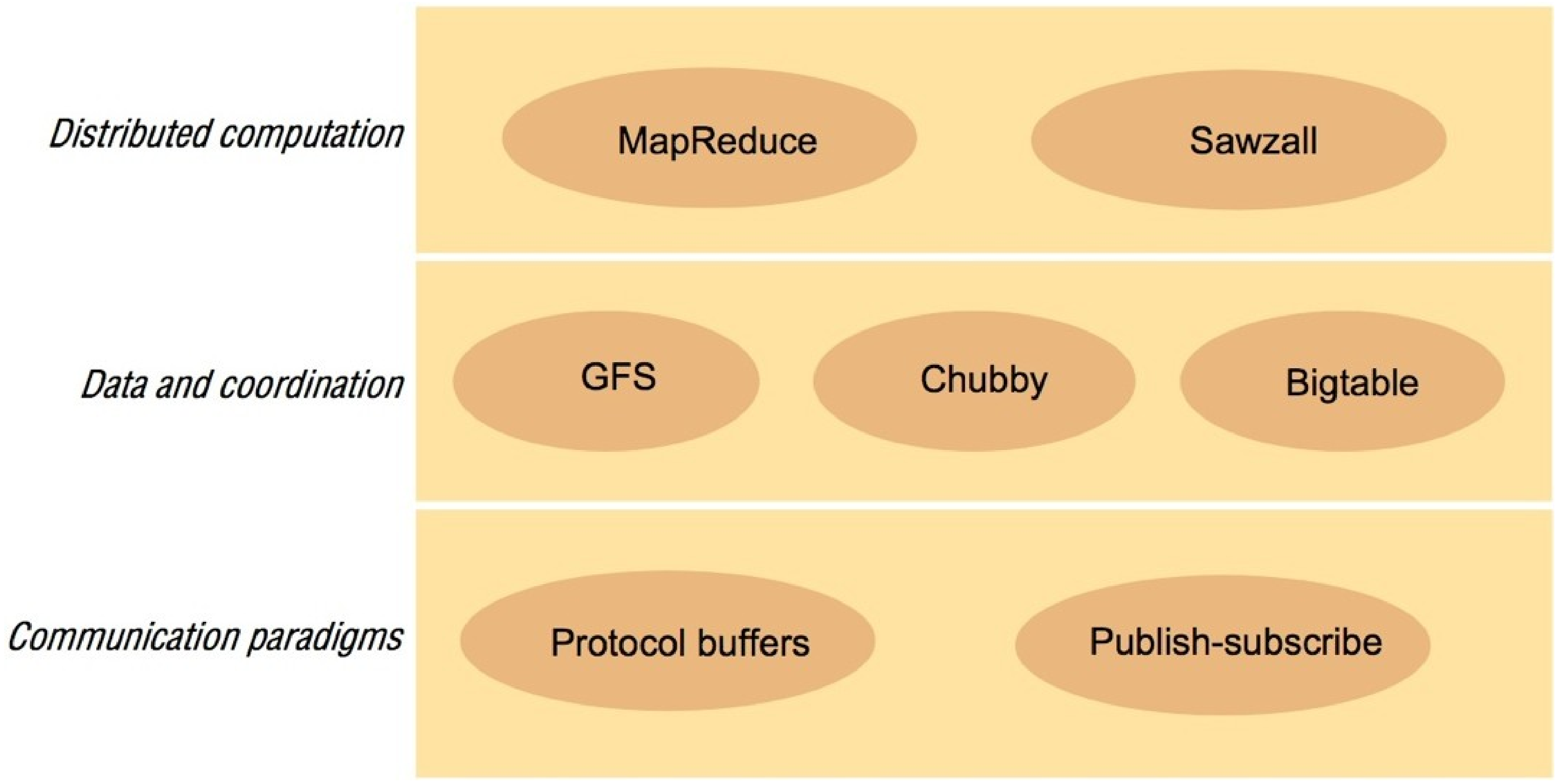
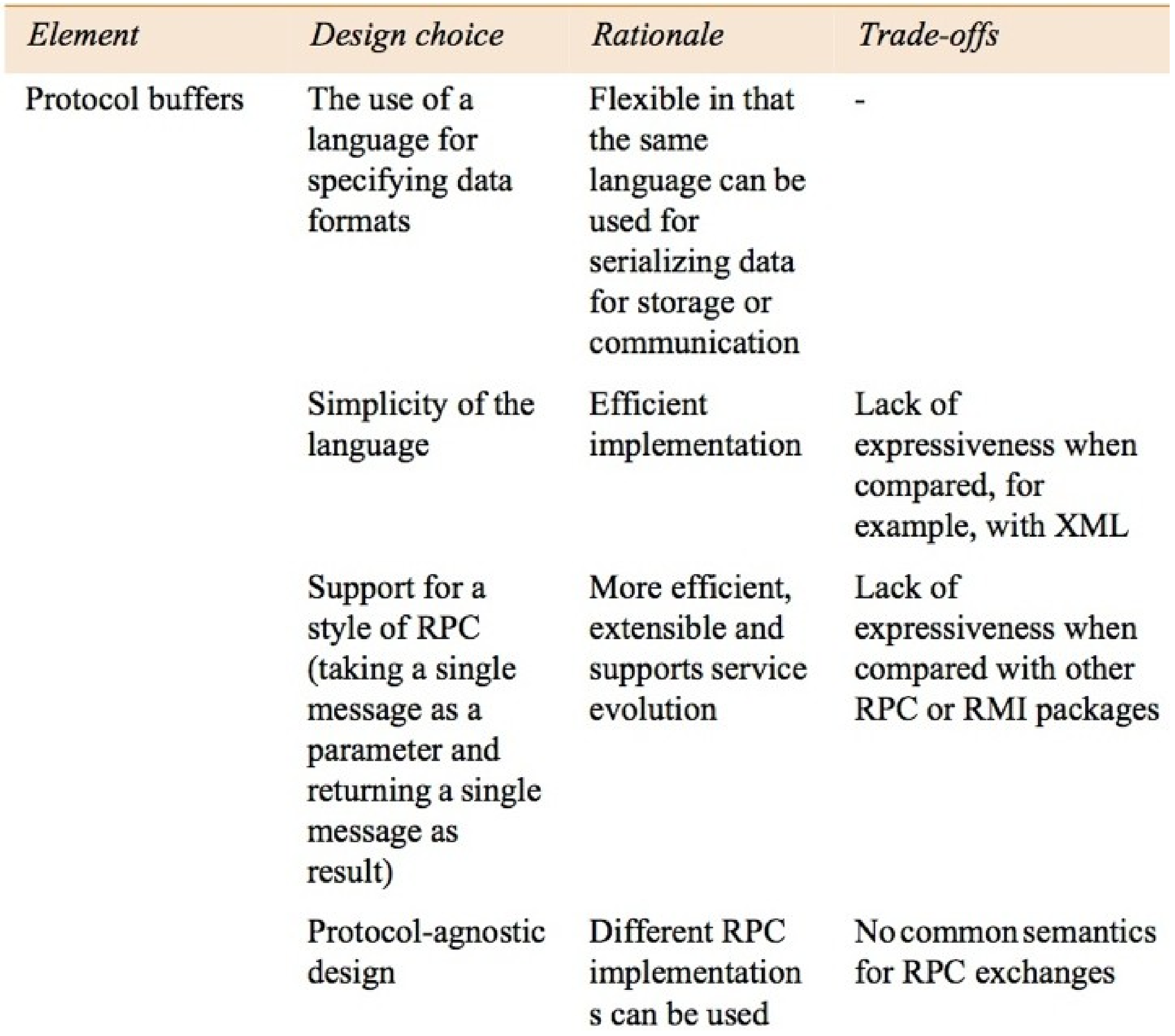


Figure 21.7
Protocol buffers example

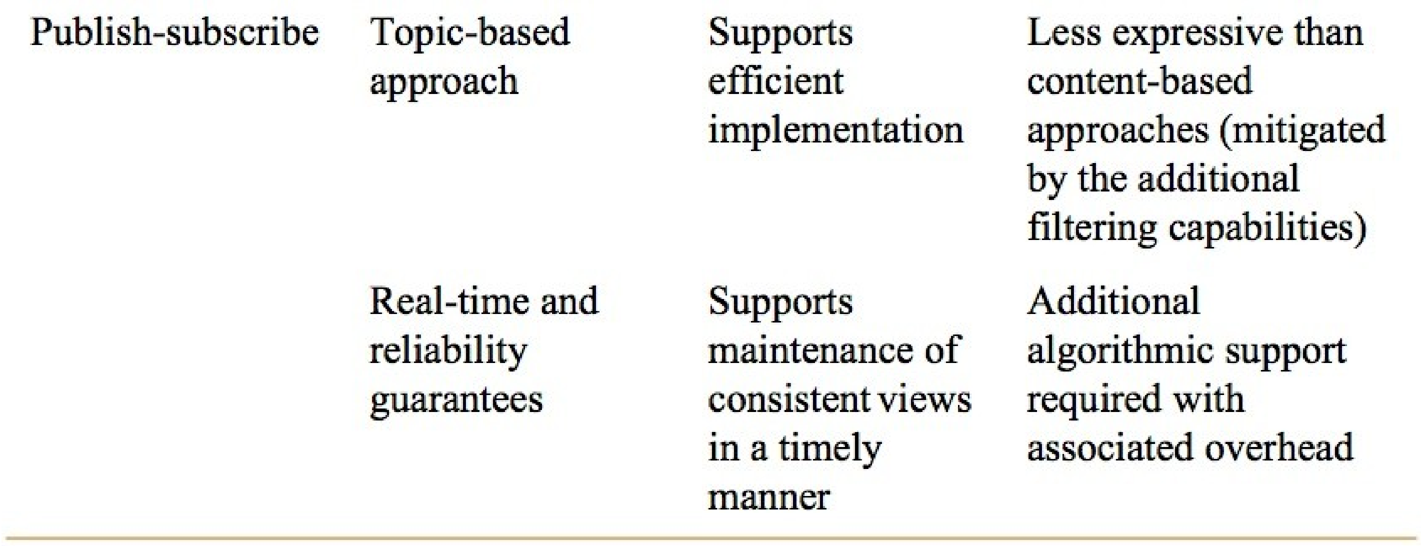
```
message Book {
    required string title = 1;
    repeated string author = 2;
    enum Status {
        IN_PRESS = 0;
        PUBLISHED = 1;
        OUT_OF_PRINT = 2;
    }
    message BookStats {
        required int32 sales = 1;
        optional int32 citations = 2;
        optional Status bookstatus = 3 [default = PUBLISHED];
    }
    optional BookStats statistics = 3;
    repeated string keyword = 4;
}
```

Figure 21.8a
Summary of design choices related to communication paradigms - *part 1*



Element	*Design choice*	*Rationale*	*Trade-offs*
Protocol buffers	The use of a language for specifying data formats	Flexible in that the same language can be used for serializing data for storage or communication	-
	Simplicity of the language	Efficient implementation	Lack of expressiveness when compared, for example, with XML
	Support for a style of RPC (taking a single message as a parameter and returning a single message as result)	More efficient, extensible and supports service evolution	Lack of expressiveness when compared with other RPC or RMI packages
	Protocol-agnostic design	Different RPC implementation s can be used	No common semantics for RPC exchanges

Figure 21.8b
Summary of design choices related to communication paradigms - *part 2*



Publish-subscribe	Topic-based approach	Supports efficient implementation	Less expressive than content-based approaches (mitigated by the additional filtering capabilities)
	Real-time and reliability guarantees	Supports maintenance of consistent views in a timely manner	Additional algorithmic support required with associated overhead

Figure 21.9
Overall architecture of GFS

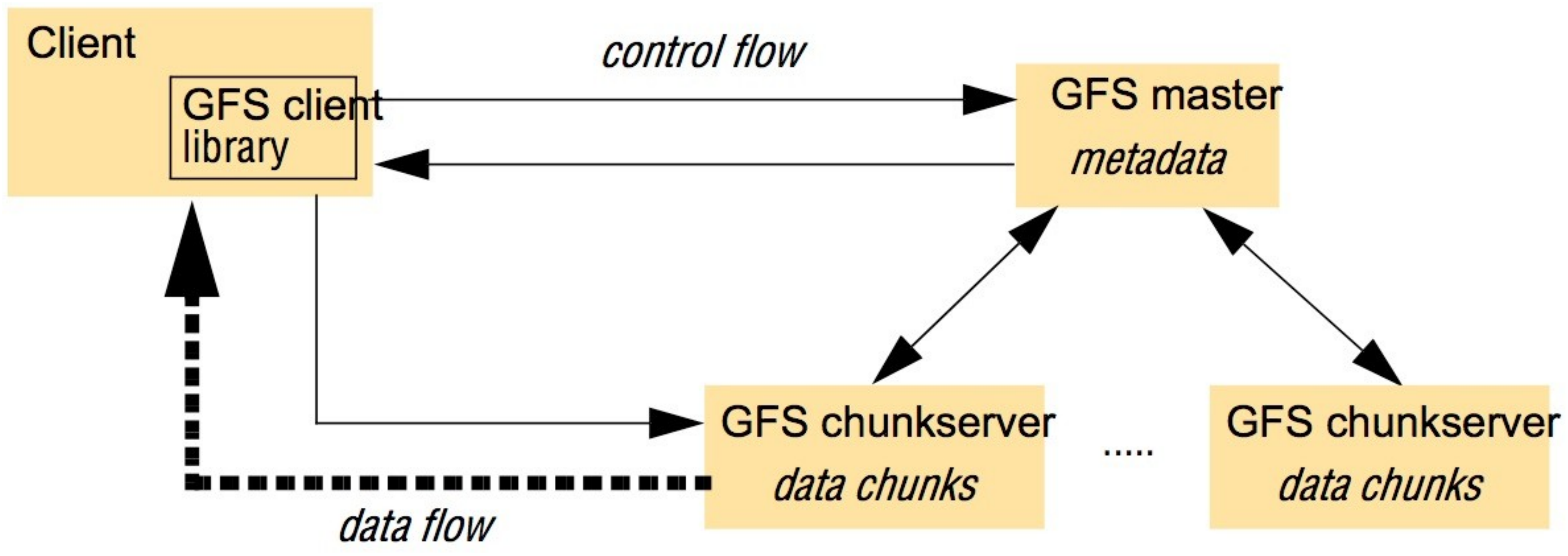
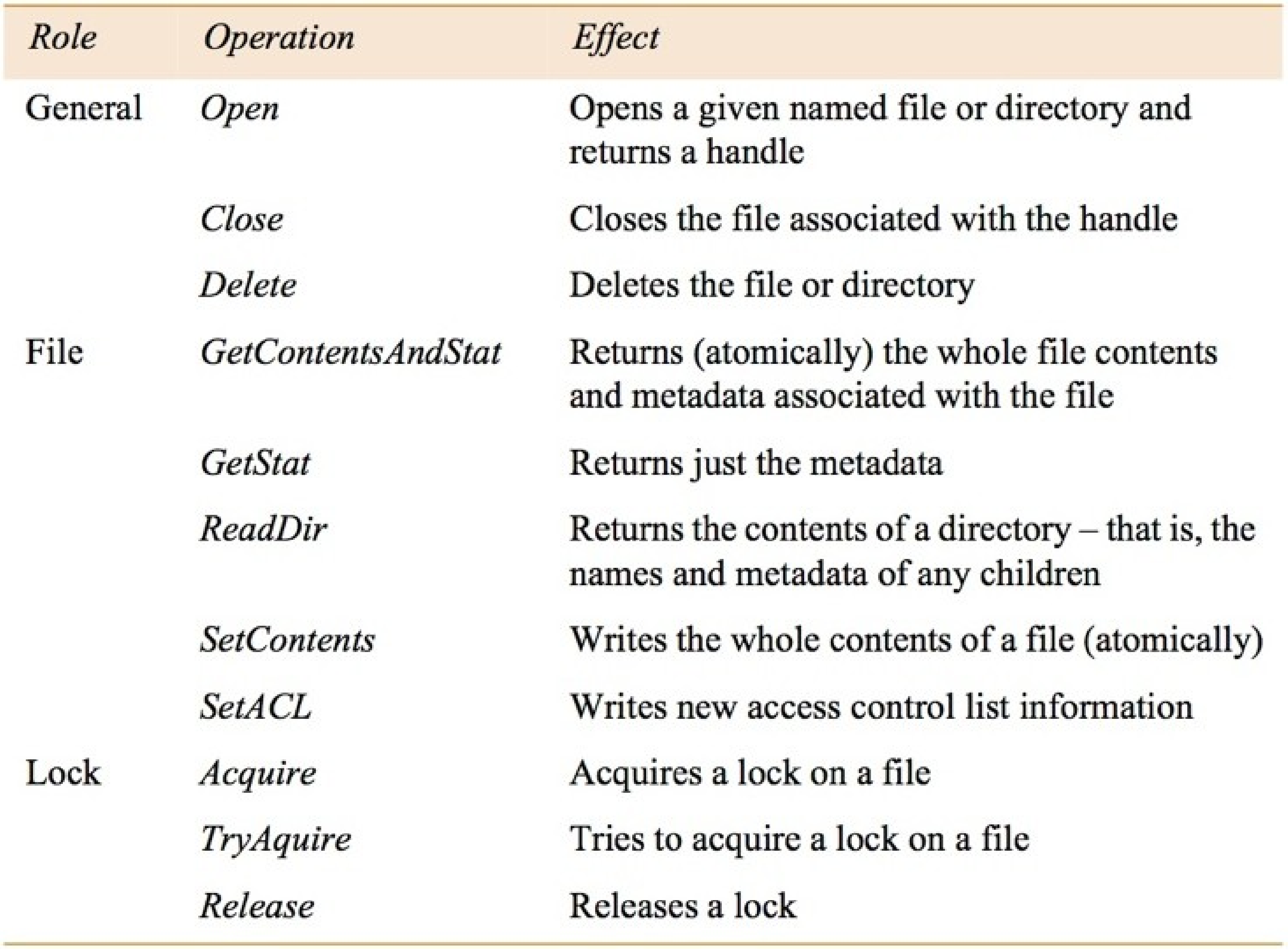


Figure 21.10
Chubby API



Role	*Operation*	*Effect*
General	*Open*	Opens a given named file or directory and returns a handle
	Close	Closes the file associated with the handle
	Delete	Deletes the file or directory
File	*GetContentsAndStat*	Returns (atomically) the whole file contents and metadata associated with the file
	GetStat	Returns just the metadata
	ReadDir	Returns the contents of a directory – that is, the names and metadata of any children
	SetContents	Writes the whole contents of a file (atomically)
	SetACL	Writes new access control list information
Lock	*Acquire*	Acquires a lock on a file
	TryAquire	Tries to acquire a lock on a file
	Release	Releases a lock

Figure 21.11
Overall architecture of Chubby

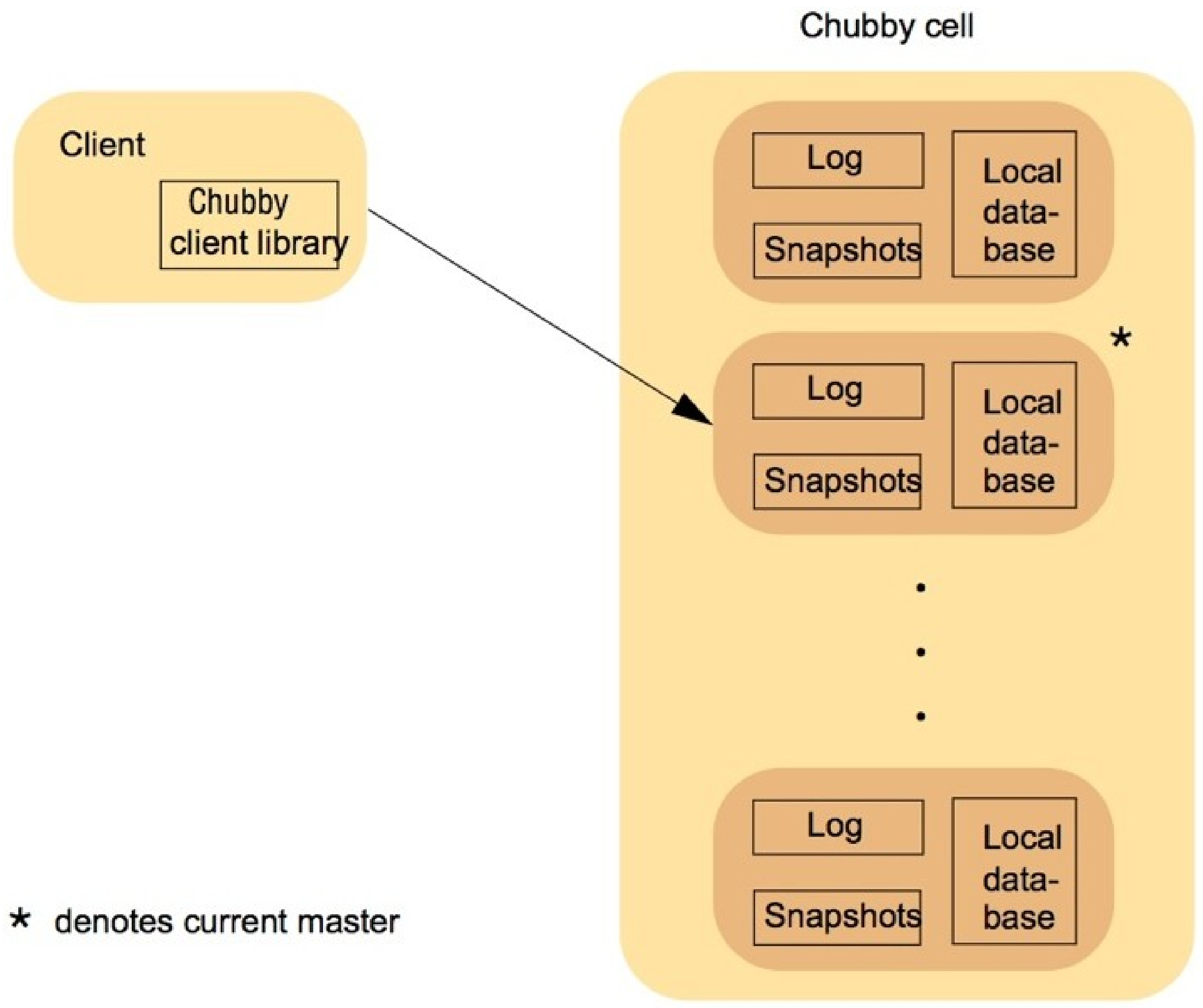


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 1

Step 1: electing a coordinator

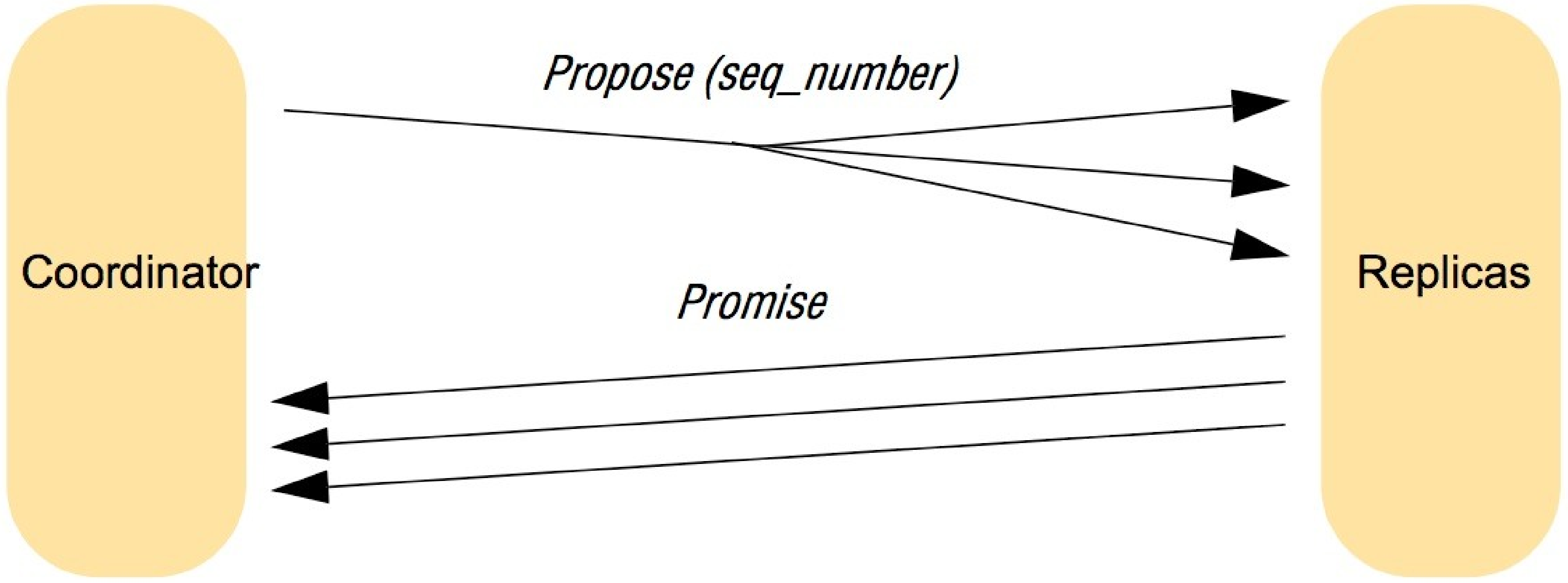


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 2

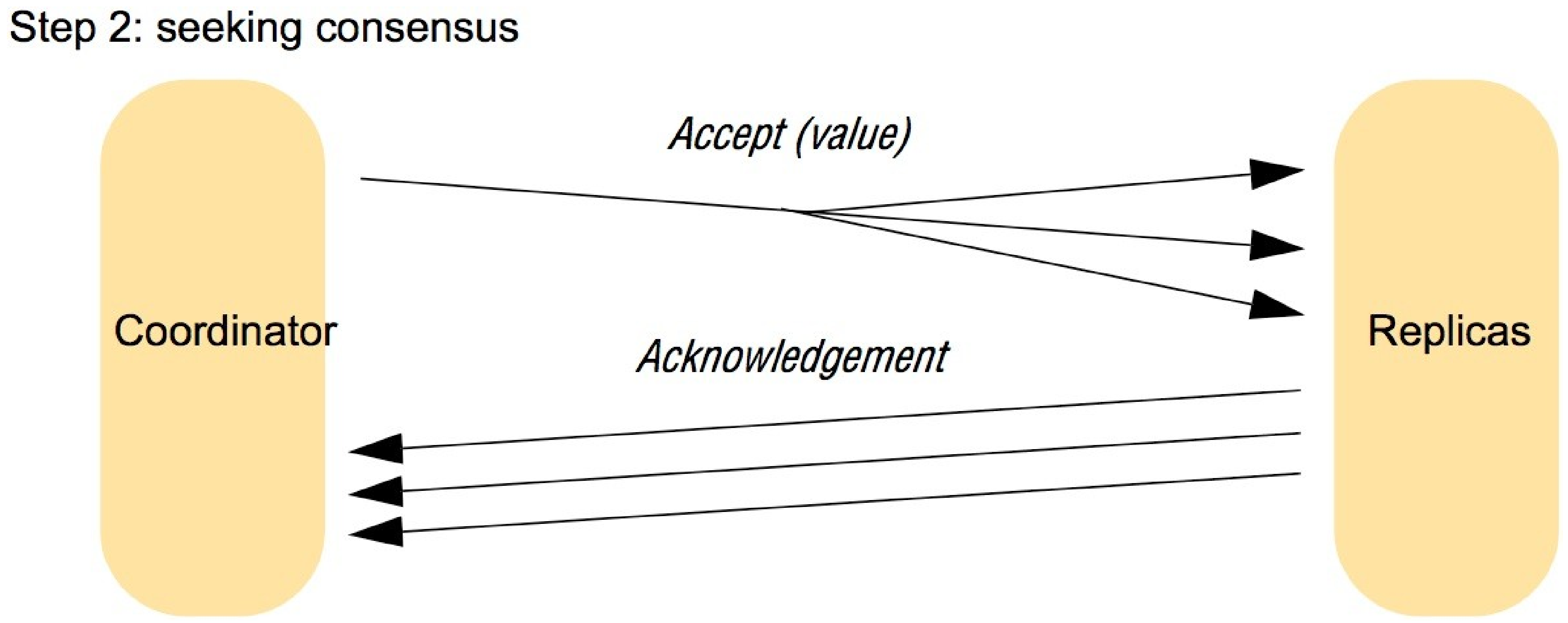


Figure 21.12
Message exchanges in Paxos (in absence of failures) - step 3

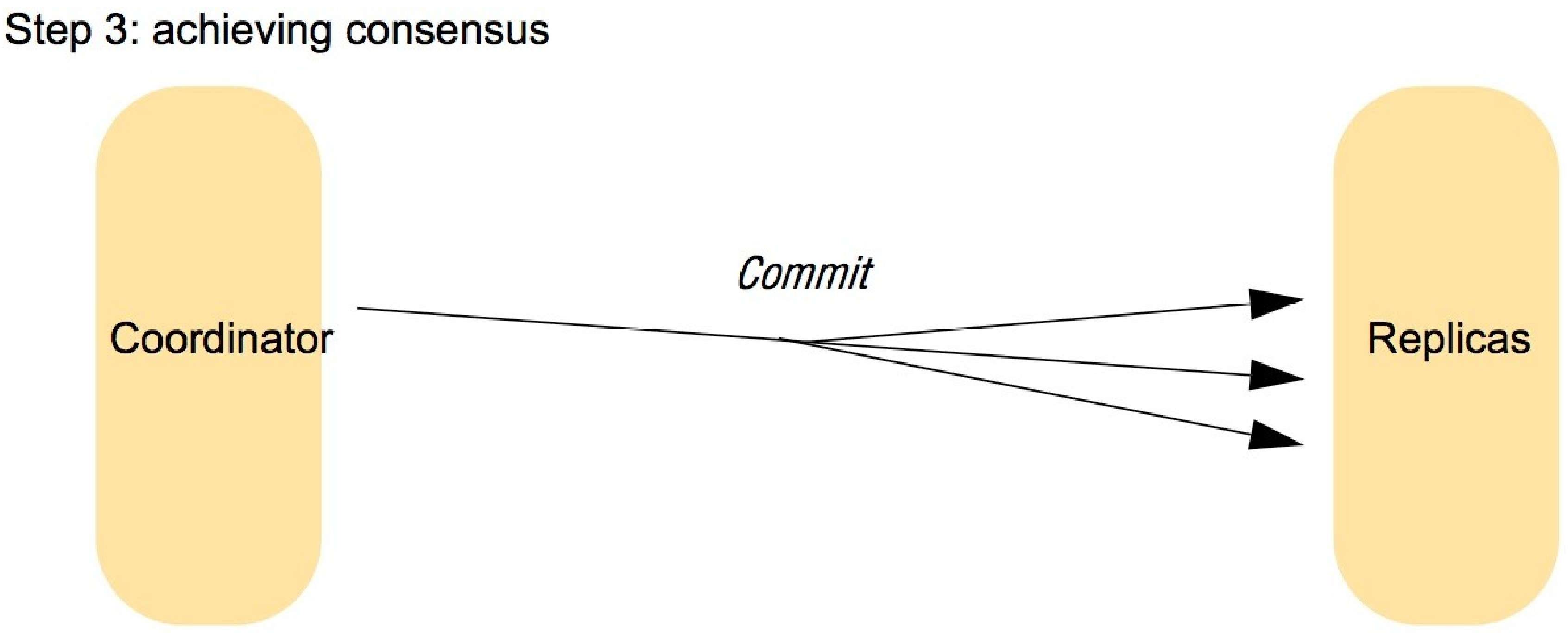


Figure 21.13
The table abstraction in Bigtable

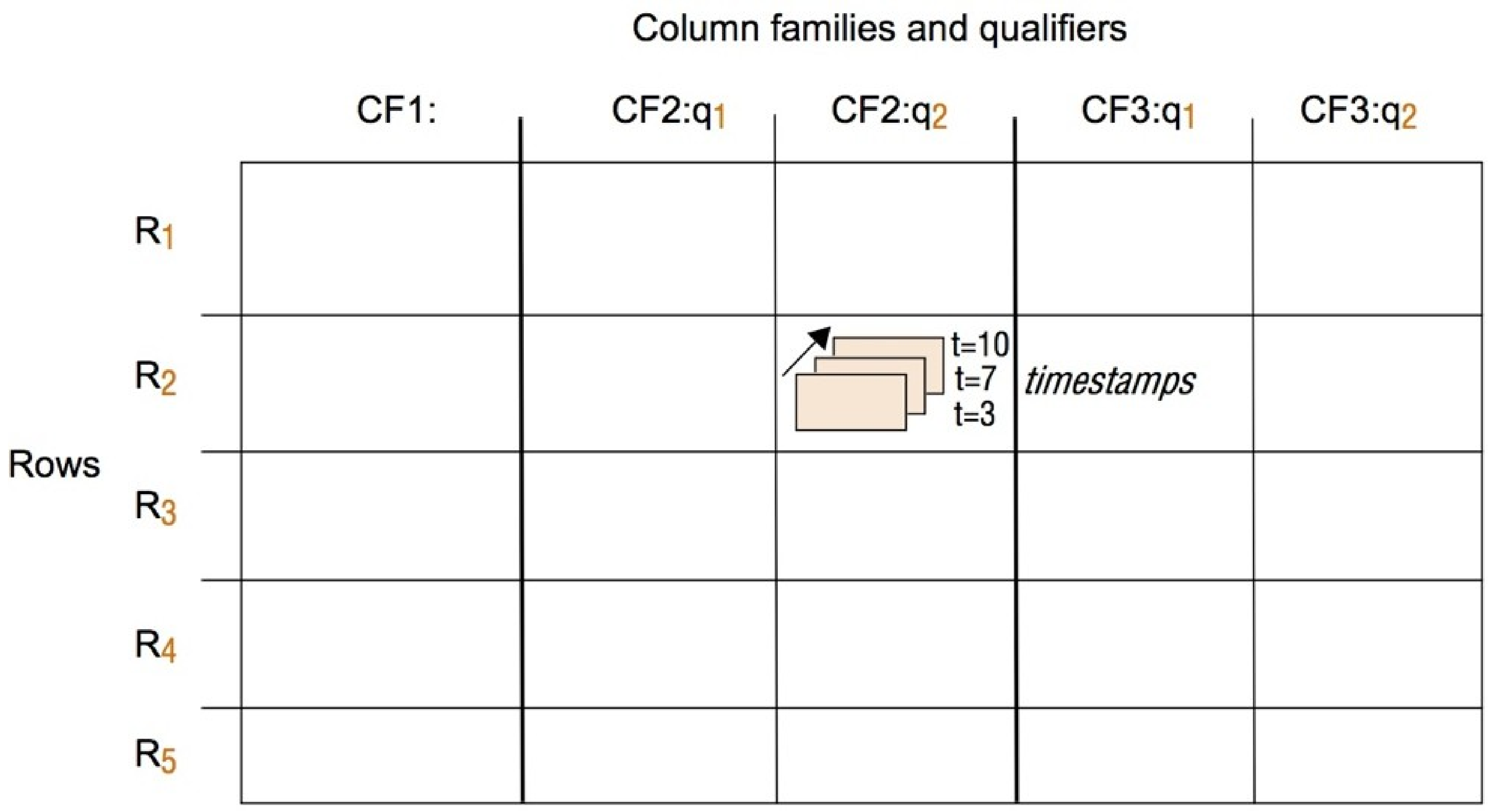


Figure 21.14
Overall architecture of Bigtable

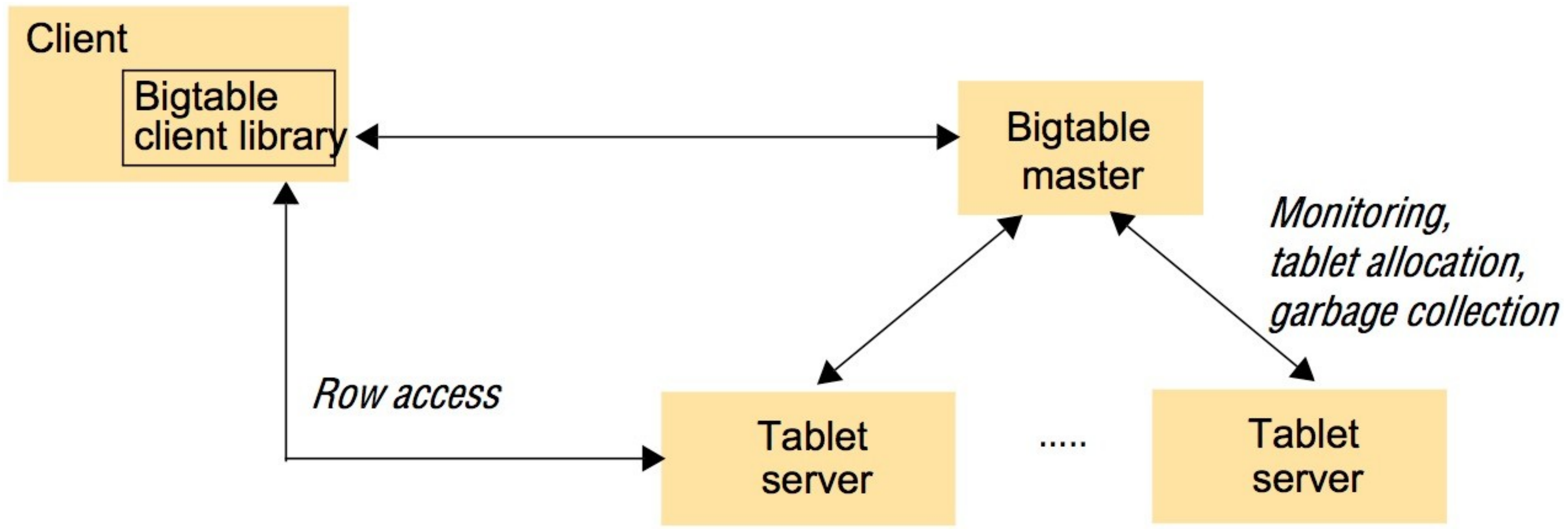


Figure 21.15
The storage architecture in Bigtable

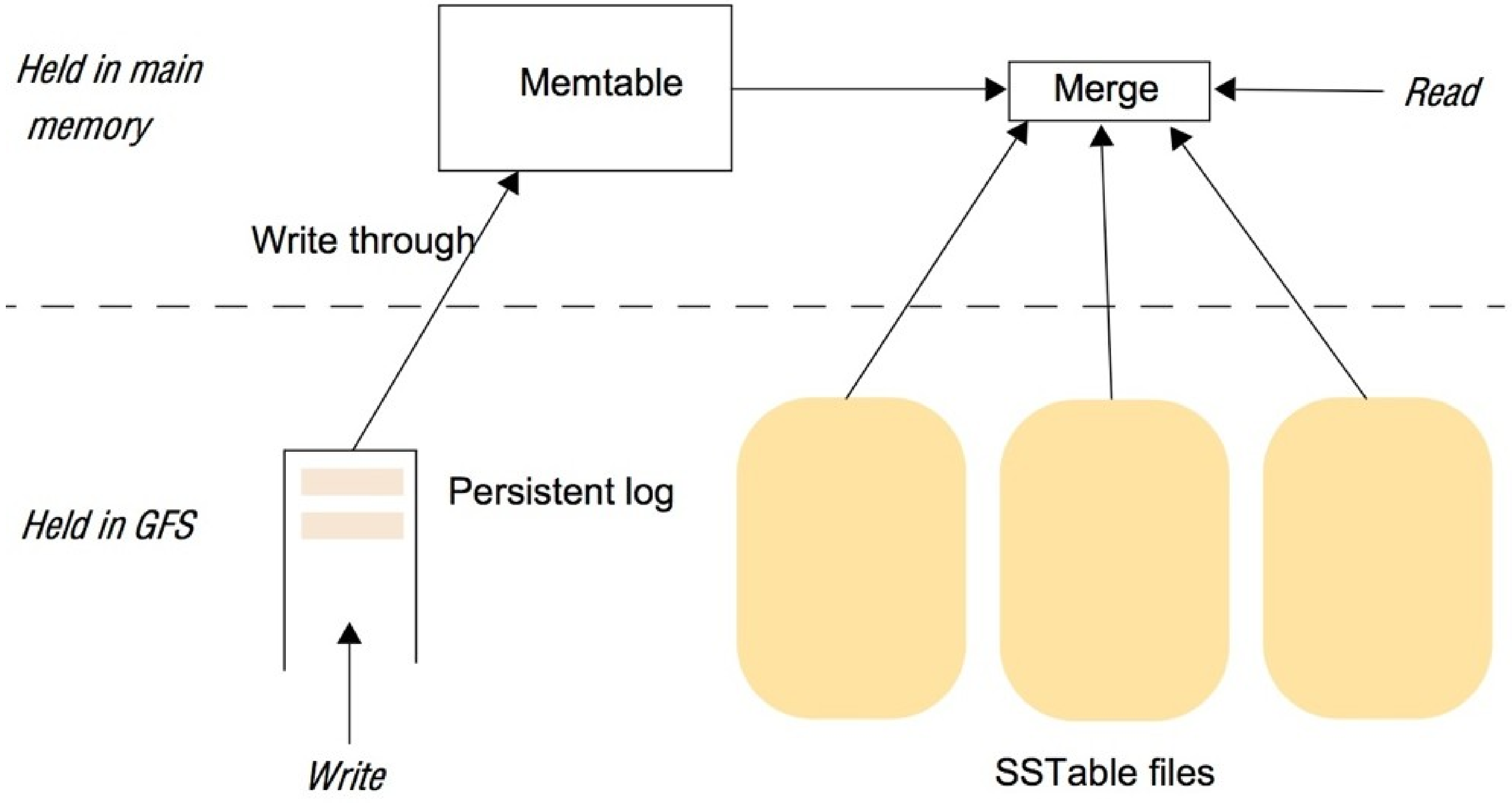


Figure 21.16
The hierarchical indexing scheme adopted by Bigtable

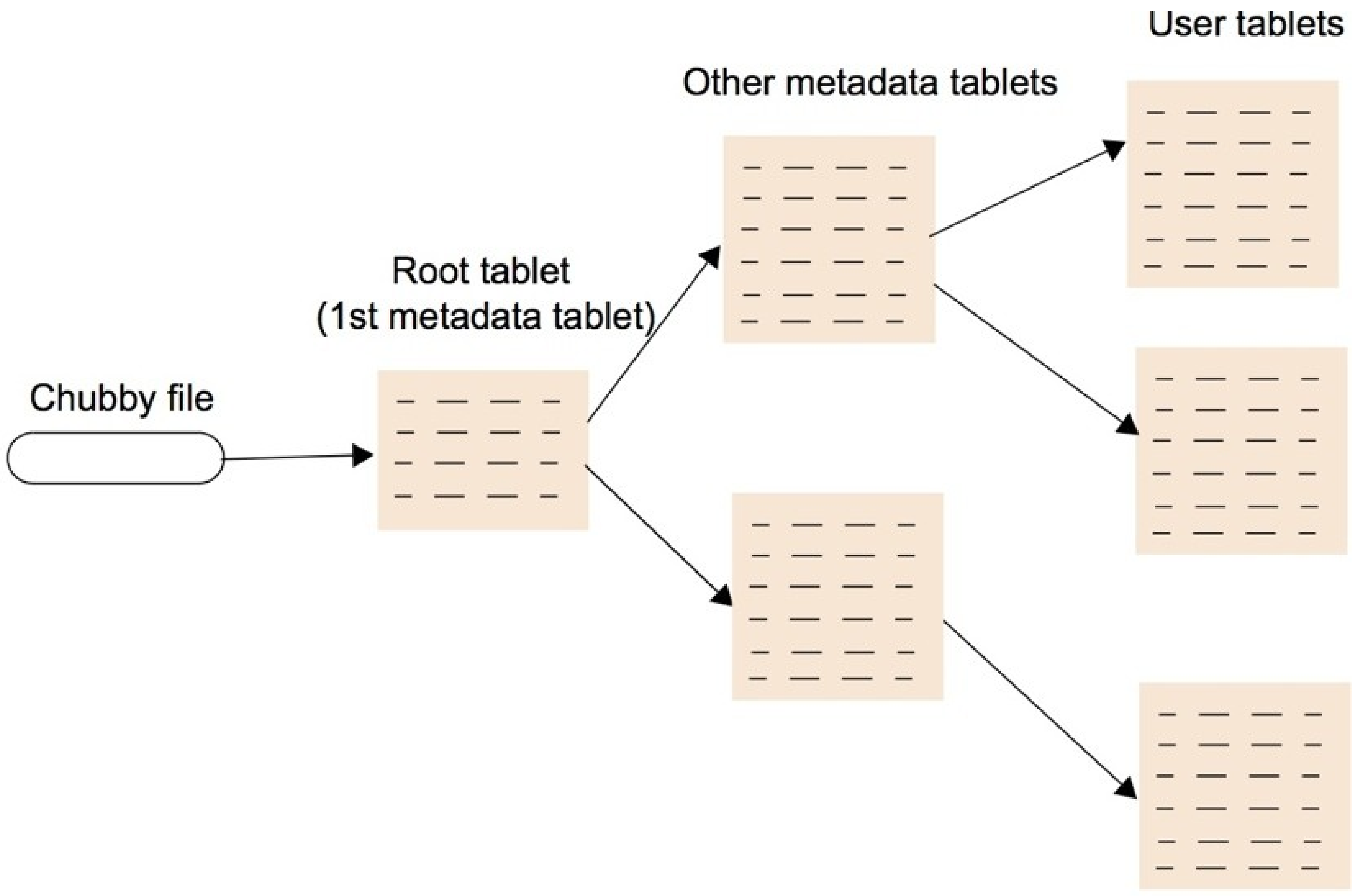
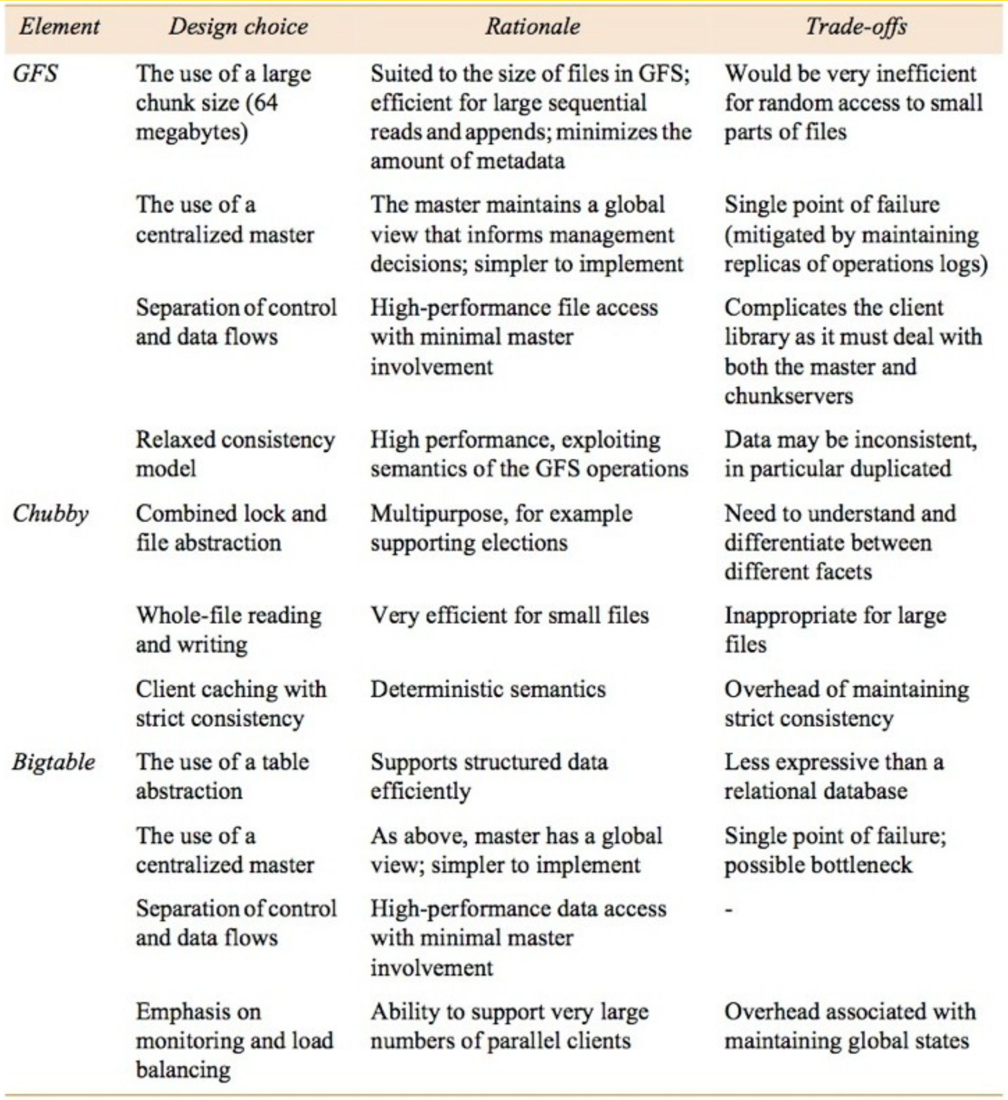
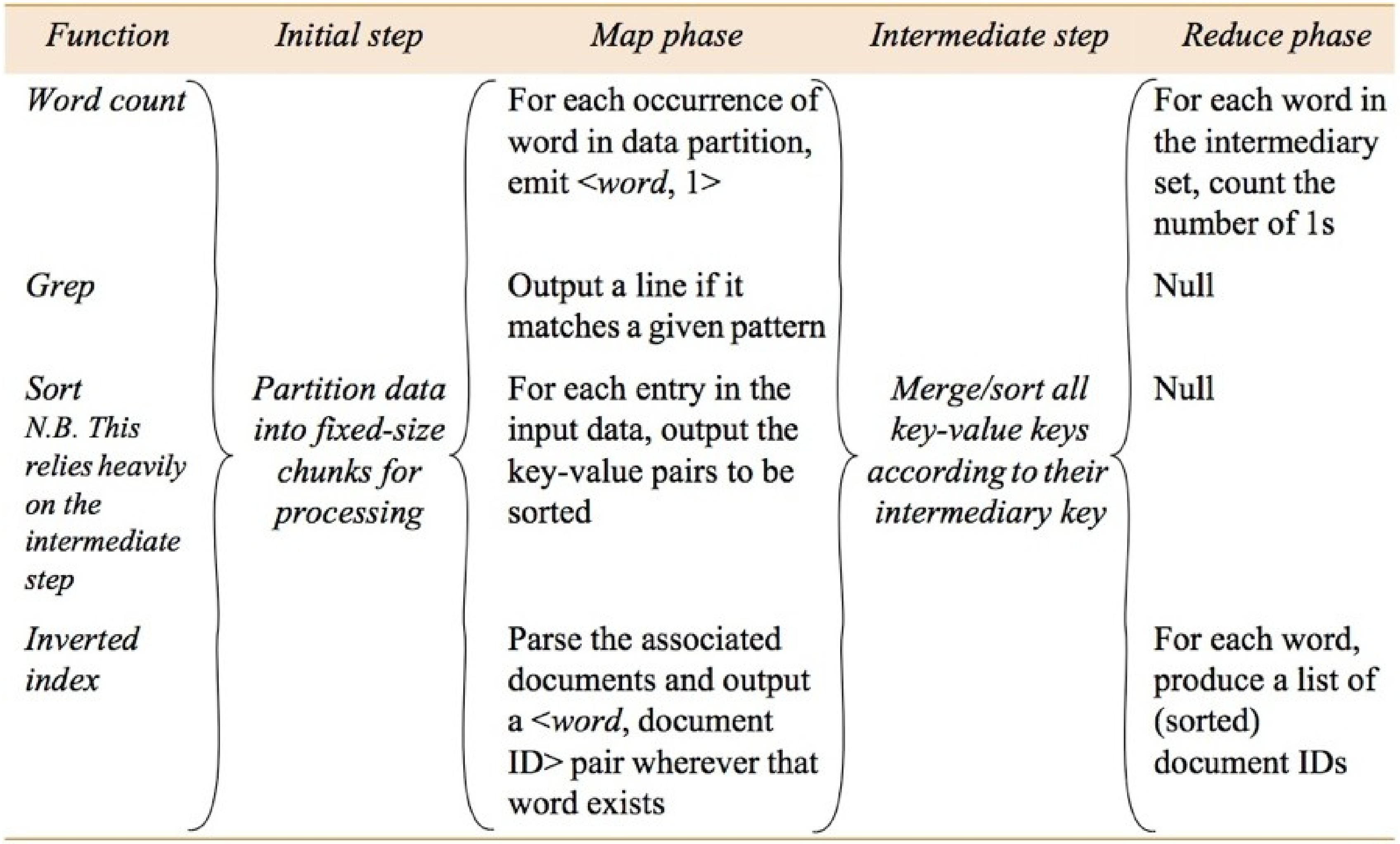


Figure 21.17
Summary of design choices related to data storage and coordination



Element	*Design choice*	*Rationale*	*Trade-offs*
GFS	The use of a large chunk size (64 megabytes)	Suited to the size of files in GFS; efficient for large sequential reads and appends; minimizes the amount of metadata	Would be very inefficient for random access to small parts of files
	The use of a centralized master	The master maintains a global view that informs management decisions; simpler to implement	Single point of failure (mitigated by maintaining replicas of operations logs)
	Separation of control and data flows	High-performance file access with minimal master involvement	Complicates the client library as it must deal with both the master and chunkservers
	Relaxed consistency model	High performance, exploiting semantics of the GFS operations	Data may be inconsistent, in particular duplicated
Chubby	Combined lock and file abstraction	Multipurpose, for example supporting elections	Need to understand and differentiate between different facets
	Whole-file reading and writing	Very efficient for small files	Inappropriate for large files
	Client caching with strict consistency	Deterministic semantics	Overhead of maintaining strict consistency
Bigtable	The use of a table abstraction	Supports structured data efficiently	Less expressive than a relational database
	The use of a centralized master	As above, master has a global view; simpler to implement	Single point of failure; possible bottleneck
	Separation of control and data flows	High-performance data access with minimal master involvement	-
	Emphasis on monitoring and load balancing	Ability to support very large numbers of parallel clients	Overhead associated with maintaining global states

Figure 21.18
Examples of the use of MapReduce



Function	*Initial step*	*Map phase*	*Intermediate step*	*Reduce phase*
Word count		For each occurrence of word in data partition, emit <*word*, 1>		For each word in the intermediary set, count the number of 1s
Grep		Output a line if it matches a given pattern		Null
Sort *N.B. This relies heavily on the intermediate step*	*Partition data into fixed-size chunks for processing*	For each entry in the input data, output the key-value pairs to be sorted	*Merge/sort all key-value keys according to their intermediary key*	Null
Inverted index		Parse the associated documents and output a <*word*, document ID> pair wherever that word exists		For each word, produce a list of (sorted) document IDs

Figure 21.19
The overall execution of a MapReduce program

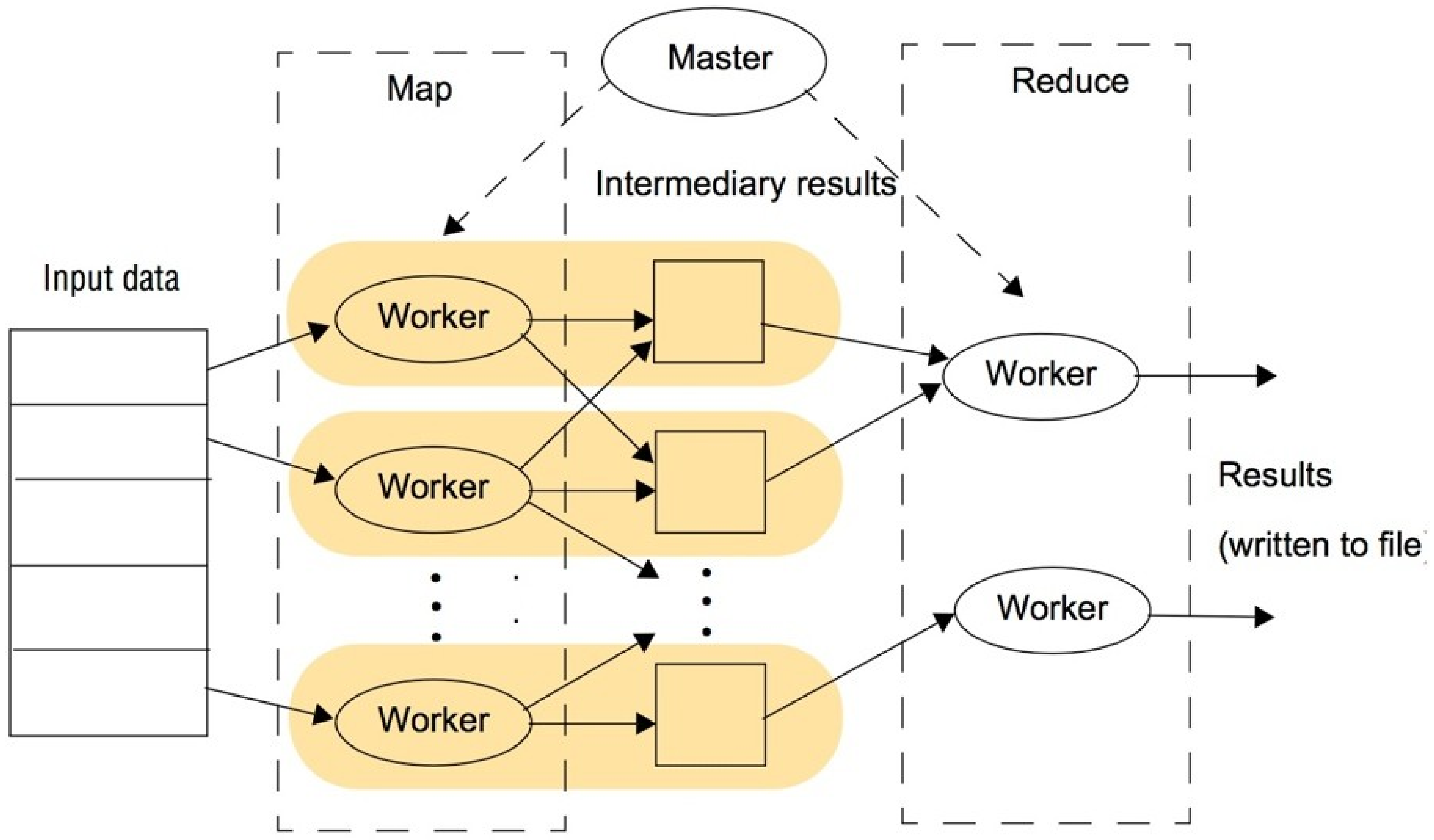


Figure 21.20
The overall execution of a Sawzall program

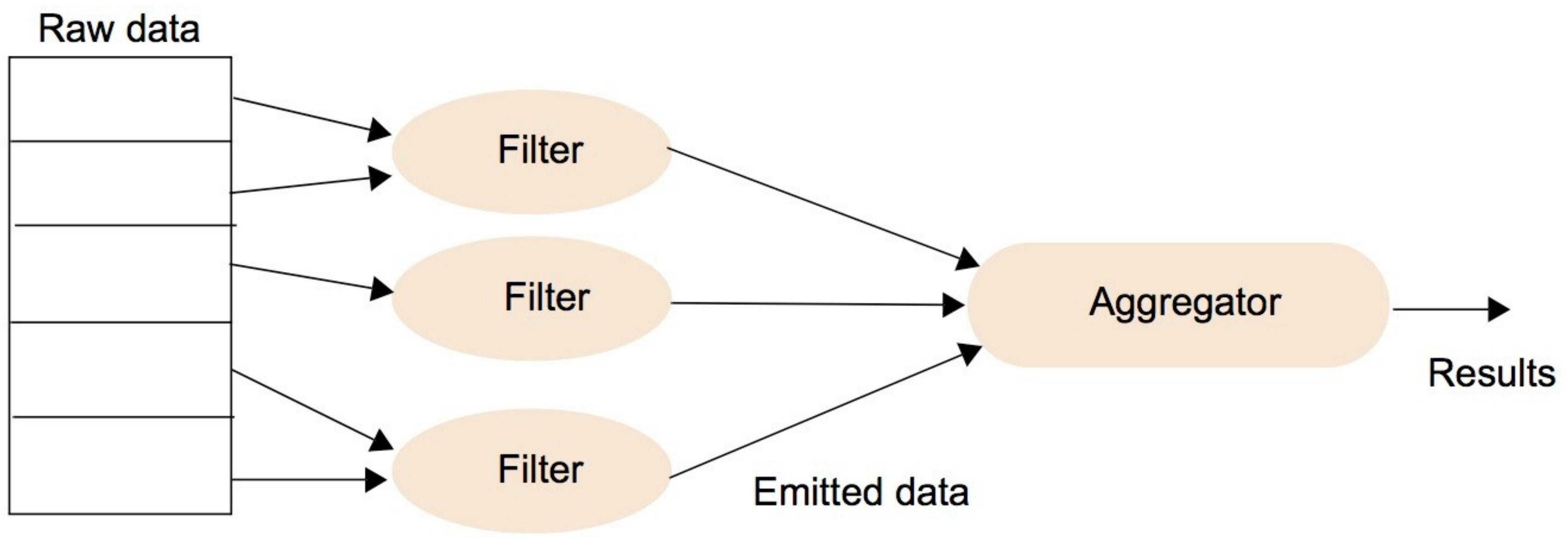
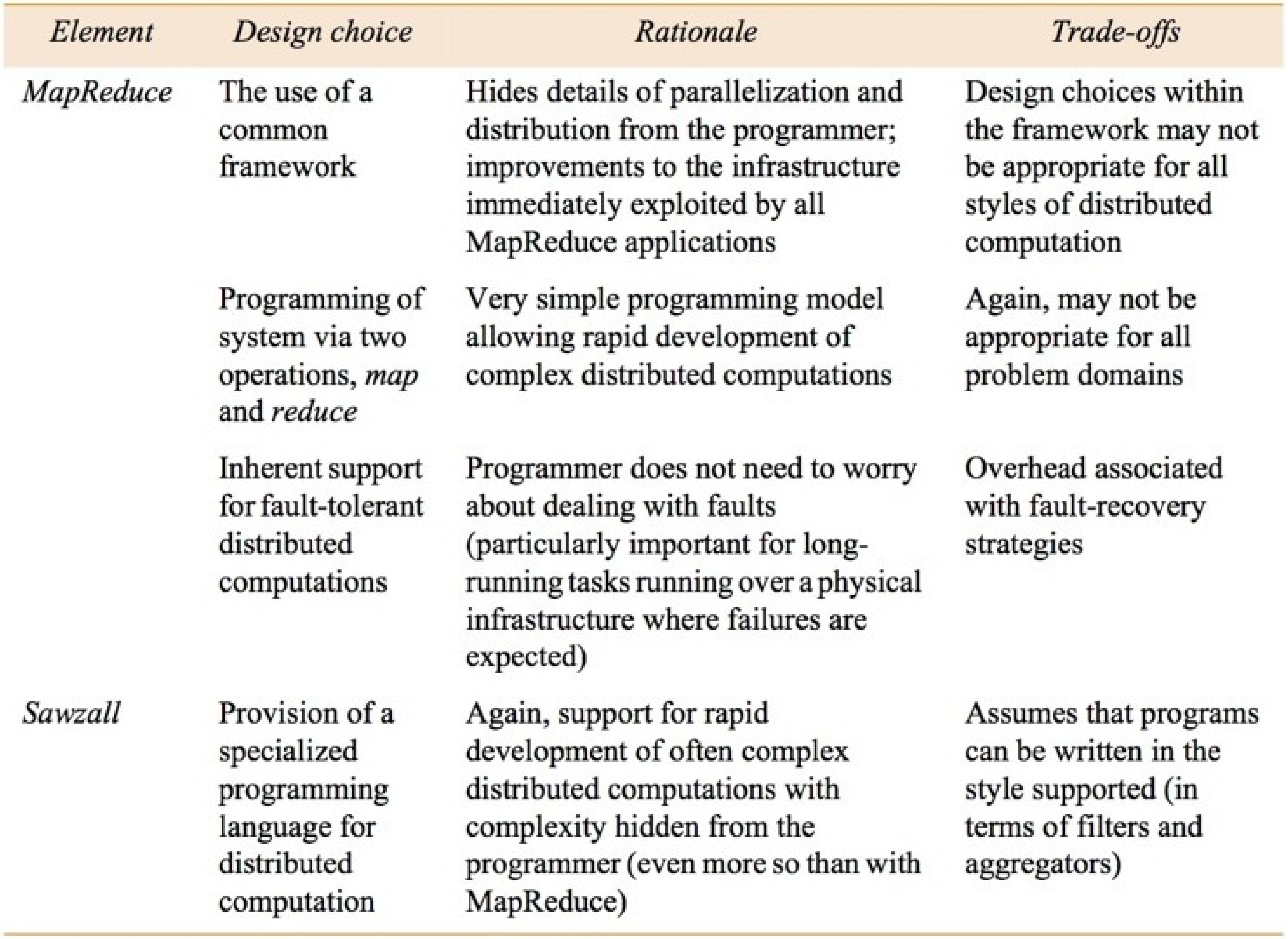


Figure 21.21
Summary of design choices related to distributed computation



Element	*Design choice*	*Rationale*	*Trade-offs*
MapReduce	The use of a common framework	Hides details of parallelization and distribution from the programmer; improvements to the infrastructure immediately exploited by all MapReduce applications	Design choices within the framework may not be appropriate for all styles of distributed computation
	Programming of system via two operations, *map* and *reduce*	Very simple programming model allowing rapid development of complex distributed computations	Again, may not be appropriate for all problem domains
	Inherent support for fault-tolerant distributed computations	Programmer does not need to worry about dealing with faults (particularly important for long-running tasks running over a physical infrastructure where failures are expected)	Overhead associated with fault-recovery strategies
Sawzall	Provision of a specialized programming language for distributed computation	Again, support for rapid development of often complex distributed computations with complexity hidden from the programmer (even more so than with MapReduce)	Assumes that programs can be written in the style supported (in terms of filters and aggregators)