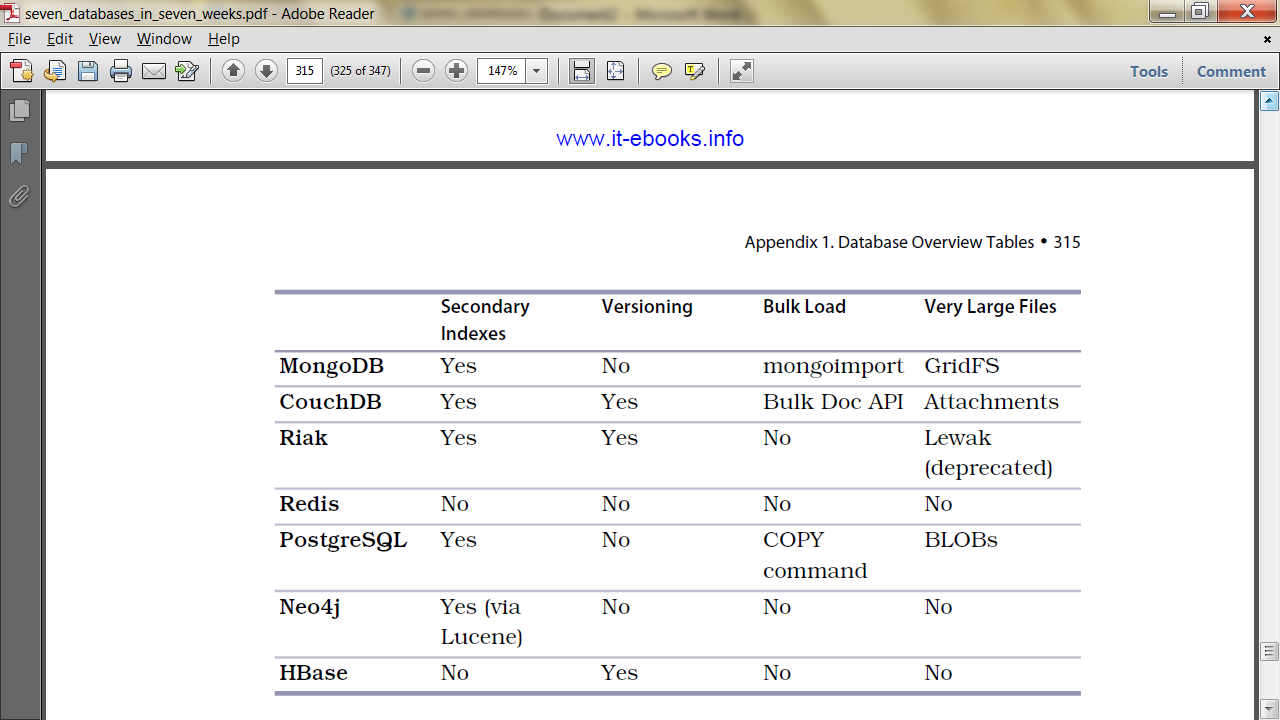
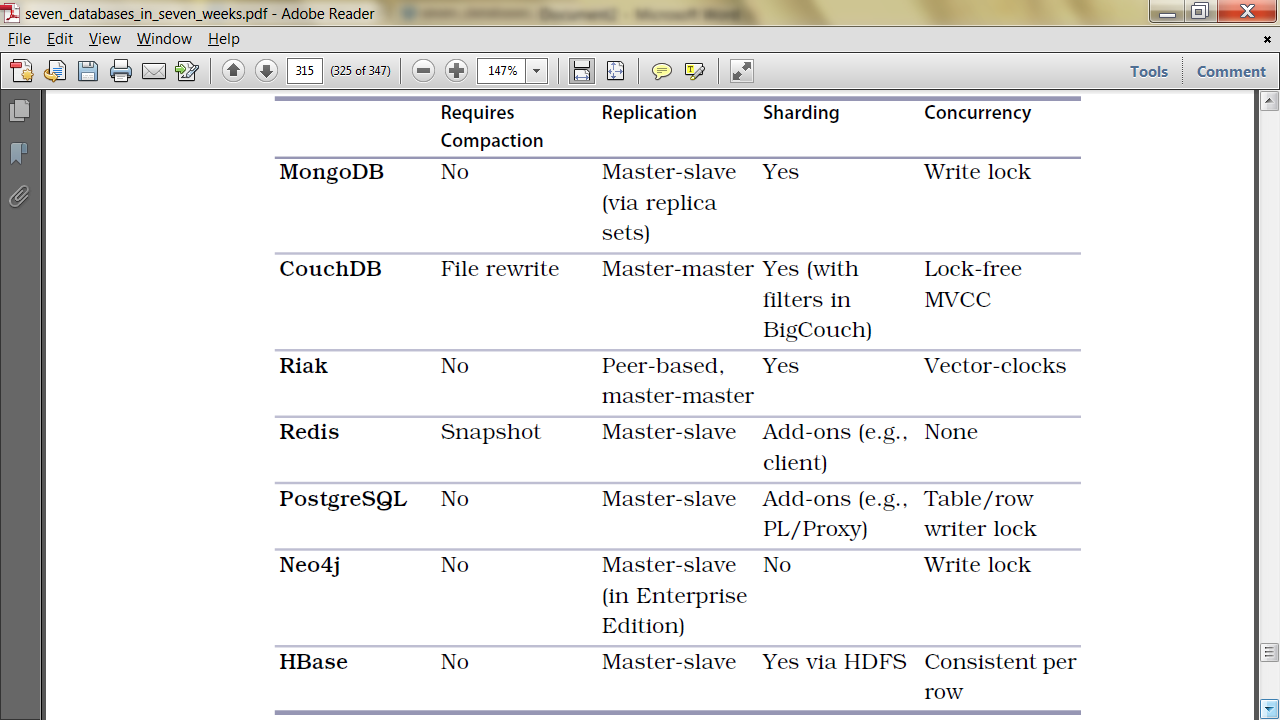


On-disk, CouchDB never overwrites committed data or associated structures, ensuring the database file is always in a consistent state. This is a “crash-only" design where the CouchDB server does not go through a shut down process, it's simply terminated. This makes the error handling quite easy: it can crash instantaneously if there is an error.

http://bladeroom.com/scalable-data-centres.php





Compaction compresses the database file by removing unused sections created during updates. Old revisions of documents are also removed from the database though a small amount of meta data is kept for use in conflict during [replication](https://wiki.apache.org/couchdb/Replication)

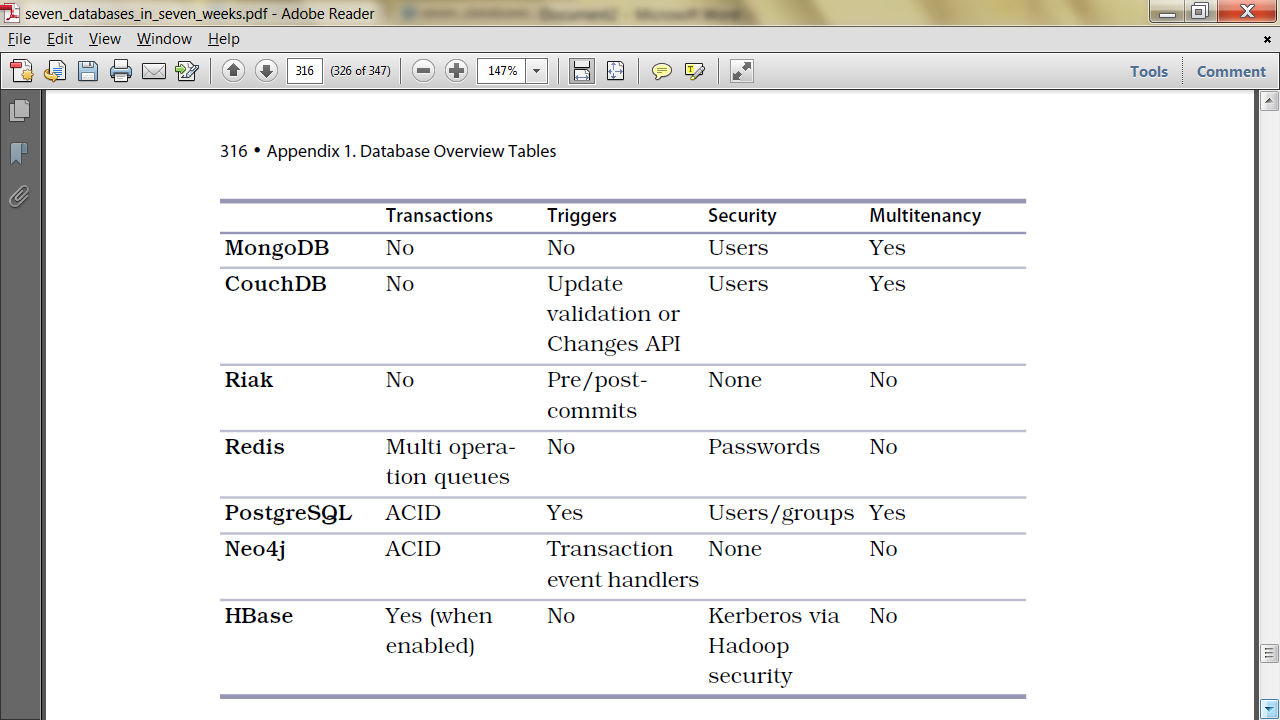
Wasted space is recovered by occasional compaction. On schedule, or when the database file exceeds a certain amount of wasted space, the compaction process clones all the active data to a new file and then discards the old file.

CouchDB's Multi-Version Concurrency Control allows us to pull revisions of our data with the attribute \_rev. Conflicts are left to the application to resolve.  Reducing bottlenecks and avoiding **locks** keeps the entire system working predictably under heavy loads.

**In HBase access to row data is atomic** and includes any number of columns being read or written to. **There is no further guarantee or transactional feature that spans multiple rows or across tables.**The atomic access is a factor to this architecture being strictly consistent, as each concurrent reader and writer can make safe assumptions about the state of a row.

When data is updated it is first written to a commit log, called a write-ahead log (WAL) in HBase, and then stored in the (sorted by RowId) in-memory memstore. Once the data in memory has exceeded a given maximum value, it is flushed as an HFile to disk. After the flush, the commit logs can be discarded up to the last unflushed modification.

**Thus a lock is needed only to protect the row in RAM.**



The term "**software multitenancy**" refers to a [software architecture](https://en.wikipedia.org/wiki/Software_architecture) in which a single [instance](https://en.wikipedia.org/wiki/Instance_(computer_science)) of [software](https://en.wikipedia.org/wiki/Computer_software) runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a [software application](https://en.wikipedia.org/wiki/Application_software) is designed to provide every tenant a dedicated share of the instance - including its data, configuration, user management, tenant individual functionality and [non-functional properties](https://en.wikipedia.org/wiki/Non-functional_requirement). Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants.

