The World's Most Widely Used Database Engine

SQLite
BerkeleyDB
LevelDB
RocksDB
etc ...

- Embedded
- Key/Value

- Embedded
- SQL

- Client/Server
- SQL
Unstoppable Ideas Behind SQL

• Transactions
  – *The system moves atomically from one consistent state to the next.*

• Data Abstraction
  – “*Representation is the essence of computer programming.*”

• Declarative Language
  – *Push the semantics of the query down into the storage engine and let it figure out what to do.*
INSERT INTO users VALUES('alex','Alexander Fogg',29,3341);

DELETE FROM users WHERE uid='alex';

UPDATE users SET officeId=4217 WHERE uid='alex';
SELECT
    blob.rid,
    uuid,
    datetime(event.mtime, 'localtime') AS timestamp,
    coalesce(ecomment, comment),
    coalesce(euser, user),
    (SELECT count(*) FROM plink WHERE pid=blob.rid AND isprim=1),
    (SELECT count(*) FROM plink WHERE cid=blob.rid),
    NOT EXISTS(SELECT 1 FROM plink
               WHERE pid=blob.rid
               AND coalesce((SELECT value FROM tagxref
                              WHERE tagid=8 AND rid=plink.pid), 'trunk')
               = coalesce((SELECT value FROM tagxref
                            WHERE tagid=8 AND rid=plink.cid), 'trunk')),
    bgcolor,
    event.type,
    (SELECT group_concat(substr(tagname,5), ', ') FROM tag, tagxref
     WHERE tagname GLOB 'sym-*' AND tag.tagid=tagxref.tagid
     AND tagxref.rid=blob.rid AND tagxref.tagtype>0),
    tagid,
    brief
FROM event JOIN blob
WHERE blob.rid=event.objid
ORDER BY event.mtime DESC LIMIT 20;
Database Files on Disk

Database Engine

Client → Database Engine
Client → Database Engine
Client → Database Engine
Client → Database Engine
Cannot connected to database
Database
Files on Disk

First code: 2000-05-29
Every Android device (~2 billion)
Every Mac and iOS device (~1 billion)
Every Win10 machine (~500 million)
Every Chrome and Firefox browser (~2 billion)
Every Skype, iTunes, WhatsApp (~2 billion)
Millions of other applications
Many billions of running instances
100s of billions, perhaps trillions, of databases
More Copies of SQLite Than...

- Linux
- Windows
- MacOS and iOS
- All other database engines combined
- Any application
- Any other library¹

¹except maybe zLib
One File Of C-code

sqlite3.c

- 205K lines
- 126K SLOC\(^1\)
- 7.2MB

Also: sqlite3.h

- 10.7K lines
- 1.5K SLOC
- 0.5MB

\(^1\)SLOC: “Source Lines Of Code” - Lines of code not counting comments and blank lines.
Open File Format

- [sqlite.org/fileformat.html](sqlite.org/fileformat.html)
- Single-file database
- Cross-platform
  - 32-bit ↔ 64-bit
  - little-endian ↔ big-endian
- Backwards-compatible
- Space efficient encoding
- Readable by 3rd-party tools
- Supported through 2050
Faster Than The File System

Time to read 100,000 BLOBs with average size of 10,000 bytes from SQLite versus directly from a file on disk.

https://sqlite.org/fasterthanfs.html
Aviation-Grade Testing

- DO-178B development process
- 100% MC/DC, as-deployed, with independence results

- Robust and reliable code - Very few bugs
- Refactor and optimize without breaking things
- Maintainable by just 3 people

https://sqlite.org/testing.html
Copyright
Storage Decision Checklist

Remote Data?

Big Data?

Concurrent Writers?

Gazillion transactions/ sec?

Otherwise

MySQL

PostgreSQL

SQL Server

Oracle

VOLTDB

SQLite
Storage Decision Checklist **FAIL!**

- Remote Data?
- Big Data?
- Concurrent Writers?
- Gazillion transactions/sec?

**No!**

Otherwise → `fopen()`
(1) Gather data from the cloud

(2) Transmit one SQLite database file to the device

(3) Use locally
CREATE TABLE sqlar( name TEXT PRIMARY KEY,  -- name of the file mode INT,       -- access permissions mtime INT,    -- last modification time sz INT,        -- original file size data BLOB  -- compressed content );

- [https://sqlite.org/sqlar](https://sqlite.org/sqlar)
- Transactional
- Concurrent & random access
- File size similar to ZIP
## SQLite versus ZIP

<table>
<thead>
<tr>
<th>Feature</th>
<th>SQLite</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container for files</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A trillion instances in the wild</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Compact</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Well-defined open format</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Container for small objects</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cross-platform small objects</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Transactions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Query language</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Schema</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Feature</td>
<td>SQLite</td>
<td>ZIP</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>Container for files</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>A trillion instances in the wild</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Compact</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Well-defined open format</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Container for small objects</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cross-platform small objects</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Transactions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Query language</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Schema</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

*The Three Unstoppable Ideas*
What If....

OpenDocument was an SQLite database
Instead of a ZIP archive of XML files...

• Fast and low-I/O save of small changes
• Fast startup
• Reduced memory usage (no need to hold the entire presentation in memory at once)
• No need for “recovery” after a crash
• No need to “file save”
• Undo across sessions
• Large searchable database of slides
SQLAR is **smaller** than ODP!

SQLAR is only 0.46% larger than ZIP
What If....

ePub was an SQLite database
Instead of a ZIP archive of XML files...

- Full text search
- Less content held in memory at once
- Faster open of large documents
- Multiple image resolutions
What if...

Git stored content in an SQLite database instead of a bespoke “packfile” key/value store?

- Ability to find descendents of check-ins
- Advanced queries for a richer user interface
- Proof against crashes
- Wiki and Tickets
- Concurrent access
- Coding errors less likely to corrupt repository
- Single-file repository
Implementation Overview

- Row-store
- Variable-length records
- Forest of B-trees
  - One B-tree for each table and each index
  - Table key: PRIMARY KEY or ROWID
  - Index key: indexed columns + table key
- Transaction control using rollback-journal or write-ahead log.
  - Atomic writes using F2FS in version 3.21.0
Ins & Outs of SQLite

SQL → Compile SQL into bytecode → Prep'ed Stmt → Bytecode Interpreter → Result

B-Tree Storage Engine
EXPLAIN SELECT price FROM tab WHERE fruit='Orange'

<table>
<thead>
<tr>
<th>addr</th>
<th>opcode</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Init</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td></td>
<td>00</td>
<td>Start at 12</td>
</tr>
<tr>
<td>1</td>
<td>OpenRead</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>00</td>
<td>root=2 iDb=0; tab</td>
</tr>
<tr>
<td>2</td>
<td>Explain</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SCAN TABLE tab 00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Rewind</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Column</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>r[1]=tab.Fruit</td>
</tr>
<tr>
<td>5</td>
<td>Ne</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td>(BINARY)</td>
<td>69</td>
<td>if r[2]!=r[1] goto 9</td>
</tr>
<tr>
<td>6</td>
<td>Column</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td></td>
<td>00</td>
<td>r[3]=tab.Price</td>
</tr>
<tr>
<td>7</td>
<td>RealAffinity</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ResultRow</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>output=r[3]</td>
</tr>
<tr>
<td>9</td>
<td>Next</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td></td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Close</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Halt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Transaction</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>IDb=0 root=2 write=0</td>
</tr>
<tr>
<td>13</td>
<td>TableLock</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>tab</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>String8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>Orange</td>
<td>00</td>
<td>r[2]='Orange'</td>
</tr>
<tr>
<td>15</td>
<td>Goto</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

Opcode documentation: [https://www.sqlite.org(opcode.html](https://www.sqlite.org/opcode.html)

Documentation generated from comments in the `vdbe.c` source file.
Two Key Objects

**sqlite3** – *Database Connection*

- An open connection to a database
- Connects to one or more databases
- May contain multiple prepared statements

**sqlite3_stmt** – *Prepared Statement*

- A single SQL statement
- Associated with a single connection
Key Methods

- sqlite3_open
- sqlite3_prepare
- sqlite3_step
- sqlite3_column
- sqlite3_finalize
- sqlite3_close
Key Methods

- sqlite3_open
- sqlite3_prepare
- sqlite3_step
- sqlite3_column
- sqlite3_finalize
- sqlite3_close
Key Methods

- `sqlite3_open`
- `sqlite3_prepare`
- `sqlite3_step`
- `sqlite3_column`
- `sqlite3_finalize`
- `sqlite3_close`
Ins & Outs

- **SQLite consists of...**
  - **Compiler** to translate SQL into byte code
  - **Virtual Machine** to evaluate the byte code

```python
sqlite3_prepare_v2()
sqlite3_step()
```
Compile SQL into bytecode 

Bytecode Interpreter 

Result 

SQL 

Compile SQL into bytecode 

Prep'ed Stmt 

Bytecode Interpreter 

Result 

Delete old B-tree storage engine 

Insert new LSM storage engine
Log Structured Merge (LSM)

**Good**
- Faster writes
- Reduced write amplification
- Linear writes
- Less SSD wear

**Bad**
- Slower reads
- Background merge process
- More space on disk
- Greater complexity
CREATE TABLE user(
    login TEXT PRIMARY KEY,
    name TEXT UNIQUE,
    officeld TEXT REFERENCES office,
    jobType TEXT REFERENCES roles,
    -- Other fields omitted....
);

INSERT INTO users(login, name, officeld, jobType)
VALUES('drh', 'Richard', '3D17', 'BDFL');

Will this be faster using LSM?
CREATE TABLE user(
    login TEXT PRIMARY KEY,
    name TEXT UNIQUE,
    officeld TEXT REFERENCES office,
    jobType TEXT REFERENCES roles,
    -- Other fields omitted....
);

INSERT INTO users(login, name, officeld, jobType)
VALUES('drh', 'Richard', '3D17', 'BDFL');

4 reads, then if everything is ok, 1 write → Slower!
CREATE TABLE user(
  login TEXT PRIMARY KEY,
  name TEXT UNIQUE,
  officeld TEXT REFERENCES office,
  jobType TEXT REFERENCES roles,
  -- Other fields omitted....
);

REPLACE INTO users(login, name, officeld, jobType)
VALUES('drh', 'Richard', '3D17', 'BDFL');
Delete old B-tree storage engine

Trendy new storage technology

- Red/Black trees in NVRam?
Key Point

Storage technology may change, but representation will continue to be the essence of computer programming.

B-tree and LSM may fade away, but SQL is likely to remain relevant.
SQL \rightarrow \text{Compile SQL into bytecode} \rightarrow \text{Prep'ed Stmt} \rightarrow \text{Bytecode Interpreter} \rightarrow \text{Result}

\text{Delete old B-tree storage engine}

\text{Some kind of column store?}
Also need to rewrite the query planner!

Delete old B-tree storage engine

Some kind of column store?
Row store vs. Column Store

- 100 columns, 10 million rows
- No indexes
- Query:

```
SELECT avg(temp3) FROM measurements;
```

Which is faster? Row-store or Column-store?
Row store vs. Column Store

- 100 columns, 10 million rows
- No indexes Index on temp3
- Query:

  ```sql
  SELECT avg(temp3) FROM measurements;
  ```

  Which is faster? Row-store or Column-store?
Row store vs. Column Store

- 100 columns, 10 million rows
- No indexes; Index on temp3 and on temp0,temp3
- Query:

  ```
  SELECT avg(temp3) FROM measurements 
  WHERE temp0 BETWEEN 50.0 AND 60.0;
  ```

Which is faster? Row-store or Column-store?
Row store vs. Column Store

- **Query** (105 columns extracted from two tables):

```sql
SELECT Z_ENT, Z_PK, Z_OPT, ZASSETCRYPTOINITIALIZATIONVECTOR, ZASSETCRYPTOTAG, ZCRYPTOINITIALIZATIONVECTOR, ZCRYPTOITERATIONCOUNT, ZCRYPTOSALT, ZCRYPTOTAG, ZCRYPTOWRAPPEDKEY, ZENCRYPTEDVALUESJSON, ZIDENTIFIER, ZISPASSWORDPROTECTED, ZMARKEDFORDELETION, ZMINIMUMSUPPORTEDNOTESVERSION, ZNEEDSFETCHFROMCLOUD, ZNEEDSTOBETCHEDFROMCLOUD, ZNEEDSTOSAVEUSERSPECIFICRECORD, ZPASSWORDHINT, ZSERVERRECORD, ZSERVERSHARE, ZUNAPPLIEDENCRIPTEDRECORD, ZUSERSPRINTCSERVERRECORD, ZZONEOWNERNAME, ZCLOUDSTATE, ZCHECKEDFORLOCATION, ZCREATIONDATE, ZDURATION, ZFILESIZE, ZHASMARKUPDATA, ZMERGEABLEDATA, ZMODIFICATIONDATE, ZORIENTATION, ZORIGINX, ZORIGINY, ZPREVIEWUPDATEDATE, ZREMOTEFILEURL, ZSECTION, ZSIZEHEIGHT, ZSIZEWIDTH, ZSUMMARY, ZTITLE, ZTYPE, ZURLSTRING, ZLOCATION, ZMEDIA, ZNOTE, ZCRYPTOMETADATAINITIALIZATIONVECTOR, ZCRYPTOMETADATATAG, ZENCRYPTEDMETADATA, ZHEIGHT, ZMETADATA, ZMODIFIEDDATE, ZSCALE, ZSCALEWHENDRAWING, ZVERSION, ZVERSIONOUTOFDATE, ZWIDTH, ZATTACHMENT, ZDEVICEIDENTIFIER, ZSTATE, ZSTATEMODIFICATIONDATE, ZACCOUNT, ZCONTENTHASHATIMPORT, ZMODIFICATIONDATEATIMPORT, ZTYPE, ZACCOUNT1, ZFILENAME, ZATTACHMENT1, ZATTACHMENTVIEWTYPE, ZCREATIONDATE1, ZFOLDERSMODIFICATIONDATE, ZINTEGERID, ZLASTVIEWEDMODIFICATIONDATE, ZLEGACYCONTENTHASHTIMPORT, ZLEGACYIMPORTDEVICEIDENTIFIER, ZLEGACYMANAGEDOBJECTIDURI, ZLEGACYMODIFICATIONDATEATIMPORT, ZLEGACYNOTEintegerID, ZLEGACYNOTEwasplainTEXT, ZMODIFICATIONDATE1, ZNOTEHASCHANGES, ZSNIPPET, ZTHUMBNAILATTACHMENTIDENTIFIER, ZTITLE1, ZACCOUNT2, ZNOTEDATA, ZACCOUNTNAMEFORACCOUNTLISTSORTING, ZSHIDDENNOTECONTAINER, ZNESTEDTITLEFORSORTING, ZSORTORDER, ZOWNER, ZACCOUNTTYPE, ZCRYPTOVERIFIER, ZDIDCHOOSEBITOMIGRATE, ZDIDFINISHMIGRATION, ZDIDMIGRATEONMAC, ZNAME, ZUSERRECORDNAME, ZDATEFORLASTTITLEMODIFICATION, ZFOLDERTYPE, ZIMPORTEDFROMLEGACY, ZPARENTMODIFICATIONDATE, ZTITLE2, ZACCOUNT3, ZPARENT FROM ZICLOUDSYNCINGOBJECT WHERE (ZNOTE IN (SELECT * FROM _Z_intarray0) AND Z_ENT = ?) ORDER BY ZNOTE;
```

Which is faster? Row-store or Column-store?
The Future Of SQLite

- Support through 2050
- 5 or 6 releases per year
- Keep it robust, backwards-compatible, free, and organic
  - Low drama
  - “It just works”
- Constantly improving the query planner AI
- Better performance
Small, Fast, Reliable
Key/Value versus Relations

Or

Throwing Shade On Git
Most recent check-in

Oldest check-in
CREATE TABLE lineage(
  parent HASH,
  child HASH,
  rank INT
);

(B,A,0)
(C,B,0)
(D,B,1)
(E,C,0)
(F,D,0)
(9,E,0)
(9,F,0)
CREATE TABLE lineage(
    parent HASH,
    child HASH,
    rank INT
);

(B,A,0)
(C,B,0)
(D,B,1)
(E,C,0)
(F,D,0)
(9,E,0)
(9,F,0)

SELECT parent FROM lineage WHERE child='E';
SELECT child FROM lineage WHERE parent='E';
Computing $n$ most recent descendents of check-in $\$root$

CREATE TABLE lineage(
    parent HASH, -- parent check-in
    child HASH, -- child check-in
    rank INT, -- 0 for primary parent
    mtime DATETIME -- time of child check-in
);

WITH RECURSIVE
    dx(id,mtime) AS ( 
        SELECT $root, mtime FROM lineage WHERE child=$root
        UNION
        SELECT lineage.child, lineage.mtime FROM dx, lineage
            WHERE lineage.parent=dx.id"
        ORDER BY 2
    )

SELECT id FROM dx LIMIT $n
Some Things Git Does Not Compute Because Of Its Use of Key/Value

- Descendents of a check-in
- Check-ins during a particular time period
- Change history for a single file
- Locate forks or detached heads
- Check-ins containing a particular version of a file
- Check-ins that touch a certain file
- First N check-ins
- List of files sorted by number of changes
- All check-ins associated with a particular user
- Check-ins before/after/around a point in time
- ... and so forth
Key/Value Limitation Leaks Into Other Aspects Of The System

- No support for tickets, wiki, documentation
- Server infrastructure is a (large) external addition (ex: GitLab)
- Large codebase dedicated to processing the bespoke “packfile” database format
- Slow response to the SHAтерred attack
- Many useful reports omitted because they are hard to implement