

From Crash Consistency to Transactions



Yige Hu
Youngjin Kwon
Vijay Chidambaram
Emmett Witchel



Persistent data is structured; crash consistency hard

- Structured data abstractions built on file system

- SQLite, BerkeleyDB... -- Embedded DB
- LevelDB, Redis, MongoDB... -- Key-value store
- Images, binary blobs... -- Files

Easy to use & deploy

- Applications manage storage themselves

- ...and poorly!
- The POSIX interface is no longer sufficient

Data safe on crash

ACID across abstractions

High performance

A transactional file system is the answer

- Structured data uses file system storage
 - Easy management often outweighs high performance
- File system transactions provides API and mechanisms

Easy to use & deploy

Data safe on crash

- Transactions preserve consistency

High performance

- Transactions reduce work & syncs
- Concurrent transactions scalable

ACID across abstractions

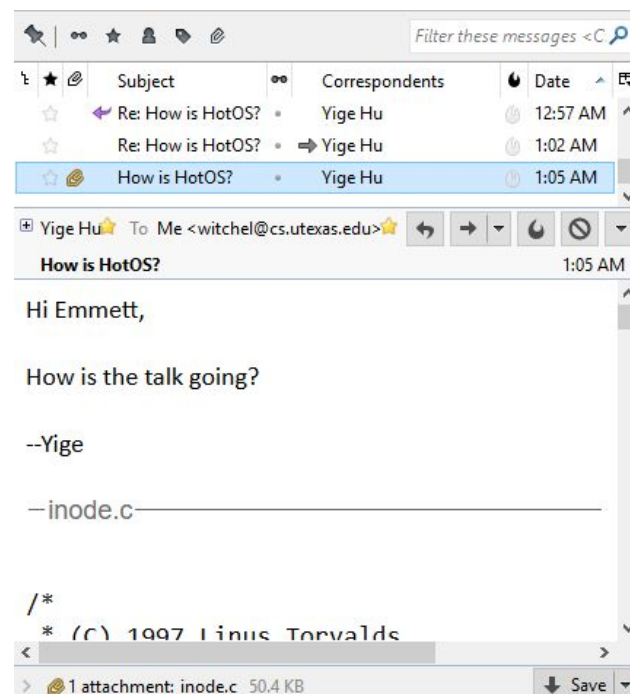
- Unify different types of updates

We need transactions across storage abstractions

- The Android mail client receives an email with attachment
 - Stores attachment as a regular file
 - File name of attachment stored in SQLite
 - Stores email text in SQLite
- Great work when you can get it, but what can go wrong?
 - Crashes can orphan attachment files
 - Crashes can leave incomplete attachments
 - **And this level of crash consistency costs dearly in performance!**

How many syncs do you need?

- The Android mail client receives an email with attachment
 - Stores attachment as a regular file (maybe 1 sync?)
 - File name of attachment stored in SQLite
 - Stores email text in SQLite (maybe 1 sync for db? 2 total?)



How many syncs do you need?

- The Android mail client receives an email with attachment
 - Stores attachment as a regular file (maybe 1 sync?)
 - File name of attachment stored in SQLite
 - Stores email text in SQLite (maybe 1 sync for db? 2?)
- **Requires 6 syncs!**
 - If you create/delete a file, sync the parent directory

Example: Android mail

Atomically inserting a message with attachment.

Raw files

SQLite

Database file

Example: Android mail

Atomically inserting a message with attachment.

Raw files

Attachment file



```
1.create(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
```

SQLite

Database file

Example: Android mail

Atomically inserting a message with attachment.

Raw files

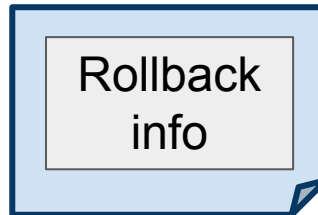
Attachment file



```
1.create(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
```

SQLite

Roll-back log



```
2.create(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
/*safe append*/
fsync(/dir/journal)
```

Database file

Example: Android mail

Atomically inserting a message with attachment.

Raw files

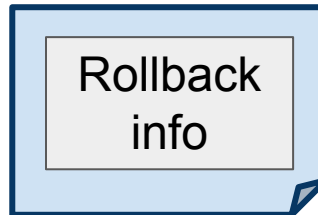
Attachment file



```
1.create(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
```

SQLite

Roll-back log



```
2.create(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
/*safe append*/
fsync(/dir/journal)
```

Database file

	/dir/attachment

```
3.write(/dir/db)
fsync(/dir/db)
```

Example: Android mail

Atomically inserting a message with attachment.

Raw files

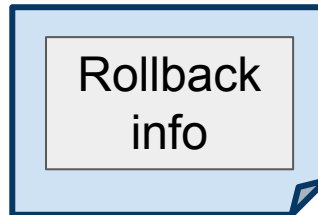
Attachment file



```
1.create(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)
```

SQLite

Roll-back log



```
2.create(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
/*safe append*/
fsync(/dir/journal)

4.unlink(/dir/journal)
```

Database file

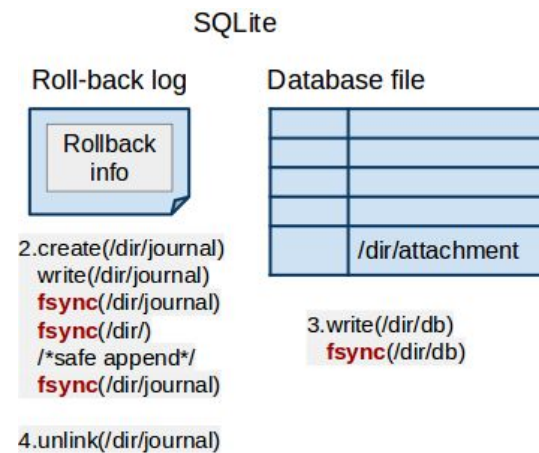
	/dir/attachment

```
3.write(/dir/db)
fsync(/dir/db)
```

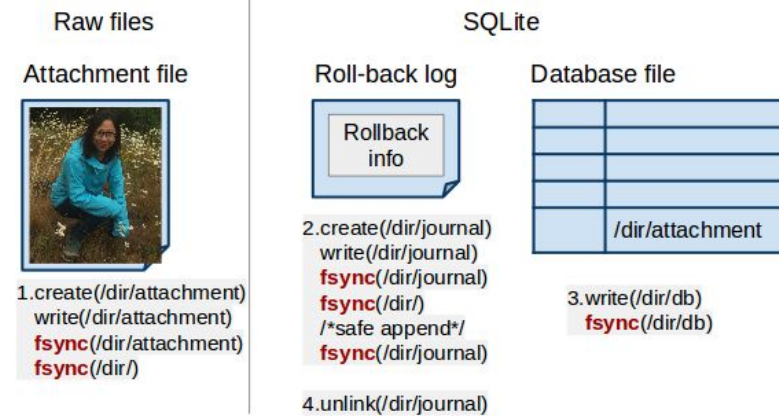
Application consistency using POSIX is slow

- SQLite on ext4: fsync() per transaction (1kB/tx), with FULL synchronization level.

	fsync/tx	
Journal mode	Insert	Update
Rollback (default)	4	4
Write ahead log (WAL)	5	5
No journal (unsafe)	1	1



System support for crash consistent updates



- Application needs consistent, persistent updates
 - Complicated and ad hoc implementation
 - Crashes can orphan attachment files
 - Crashes can create incomplete attachment files.
- Sync and redundant writes lead to poor performance.
- Need mechanism for cross-abstraction commit

The file system should provide transactional services!

But haven't we tried this before?

Haven't we seen this movie before?

- Complex implementation
 - Transactional OS: QuickSilver [TOCS 88], TxOS [SOSP 09] (**10k LOC**)
 - In-kernel transactional file systems: Valor [FAST 09]
- Hardware dependent
 - CFS [ATC 15], MARS [SOSP 13], TxFLash [OSDI 08], Isotope [FAST 16]
- Performance overhead
 - Valor [FAST 09] (**35% overhead**).
- Hard to use
 - Windows NTFS (TxF), released 2006 (deprecated 2012)

Windows TxF was hard to use

Modify the following code to use **Windows NTFS (TxF)** transactions.

```
HANDLE hFile = CreateFile(_T("test.file"),
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFile failed" << endl;
    return 1;
}

CloseHandle(hFile);
```

Windows TxF was hard to use

Modify the following code to use

```
HANDLE hFile = CreateFile(_T("test.file"),
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFile failed" << endl;
    return 1;
}

CloseHandle(hFile);
```



```
#include <ktmw32.h>
#pragma comment(lib, "KtmW32.lib")

.....
HANDLE hTrans = CreateTransaction(NULL, 0, 0, 0, 0, NULL,
    _T("My NTFS Transaction"));
if (hTrans == INVALID_HANDLE_VALUE)
{
    cerr << "CreateTransaction failed" << endl;
    return 1;
}

USHORT view = 0xFFFFE; // TXFS_MINIVERSION_DEFAULT_VIEW
HANDLE hFile = CreateFileTransacted(_T("test.file"),
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0,
    hTrans, &view, NULL);
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFileTransacted failed" << endl;
    return 1;
}

CloseHandle(hFile);

CommitTransaction(hTrans);
CloseHandle(hTrans);
```

Windows TxF was hard to use

Modify the following code to use

```
HANDLE hFile = CreateFile(_T("test.file"),
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFile failed" << endl;
    return 1;
}

CloseHandle(hFile);
```



```
#include <ktmw32.h>
#pragma comment(lib, "KtmW32.lib")
```

```
.....
HANDLE hTrans
_T("My NTFS
if (hTrans ==
{
    cerr <<
    return
}
```

```
USHORT view
HANDLE hFile
GENERIC
hTrans
if (hFile == IN
{
    cerr <<
    return
}
```

```
CloseHandle
```

```
CommitTransaction(m
CloseHandle(hTrans);
```

GetFileAttributesTransacted
CopyFileTransacted
DeleteFileTransacted

.....
**+ 16 new transactional file
operations**

Windows TxF was hard to use

Modify the

```
HANDLE hFile = Cr  
GENERIC_WRITE, C  
if (hFile == INVALID  
{  
    cerr << "Cre  
    return 1;  
}  
  
CloseHandle(hFile
```

- Microsoft deprecates TxF (2012)

“While TxF is a powerful set of APIs, there has been extremely limited developer interest in this API platform since Windows Vista primarily **due to its complexity and various nuances** which developers need to consider as part of application development.”

```
return  
}  
CloseHandle  
  
CommitTransaction(m  
CloseHandle(hTrans);
```

T2FS (Texas Transactional File System)

- Based on Linux ext4
 - Uses file system journal **Data safe on crash**
- Simple interface
 - `fs_tx_begin`, `fs_tx_end`, `fs_tx_abort` **Easy to use & deploy**
- Usable by any abstraction that stores data in the file system
 - E.g., embedded databases, key-value stores **ACID across abstractions**
- Improves performance for structured data
 - Fewer sync calls **High performance**
- Increases scalability

T2FS API

Modify the following code to use T2FS transactions.

```
HANDLE hFile = CreateFile(_T("test.file"),
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFile failed" << endl;
    return 1;
}

CloseHandle(hFile);
```

T2FS API

Modify the following code to use T2FS transactions.

```
fs_tx_begin();
```

```
HANDLE hFile = CreateFile(_T("test.file"),  
    GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);  
if (hFile == INVALID_HANDLE_VALUE)  
{  
    cerr << "CreateFile failed" << endl;  
    return 1;  
}
```

```
CloseHandle(hFile);
```

```
fs_tx_end();
```

T2FS API

Modify the following code to

fs_tx_begin();

```
HANDLE hFile = CreateFile(
    GENERIC_READ,
    if (hFile == INVALID_HANDLE_VALUE)
    {
        cerr << "CreateFile failed" << endl;
        return 1;
    }
```

CloseHandle(hFile);

fs_tx_end();

T2FS API

```
#include <ktmw32.h>
#pragma comment(lib, "KtmW32.lib")

.....
HANDLE hTrans = CreateTransaction(NULL, 0, 0, 0, 0, NULL,
    _T("My NTFS Transaction"));
if (hTrans == INVALID_HANDLE_VALUE)
{
    cerr << "CreateTransaction failed" << endl;
    return 1;
}
```

USHORT view = 0xFFFFE; // TXFS_MINIVERSION_DEFAULT_VIEW

Windows NTFS (TxF) API

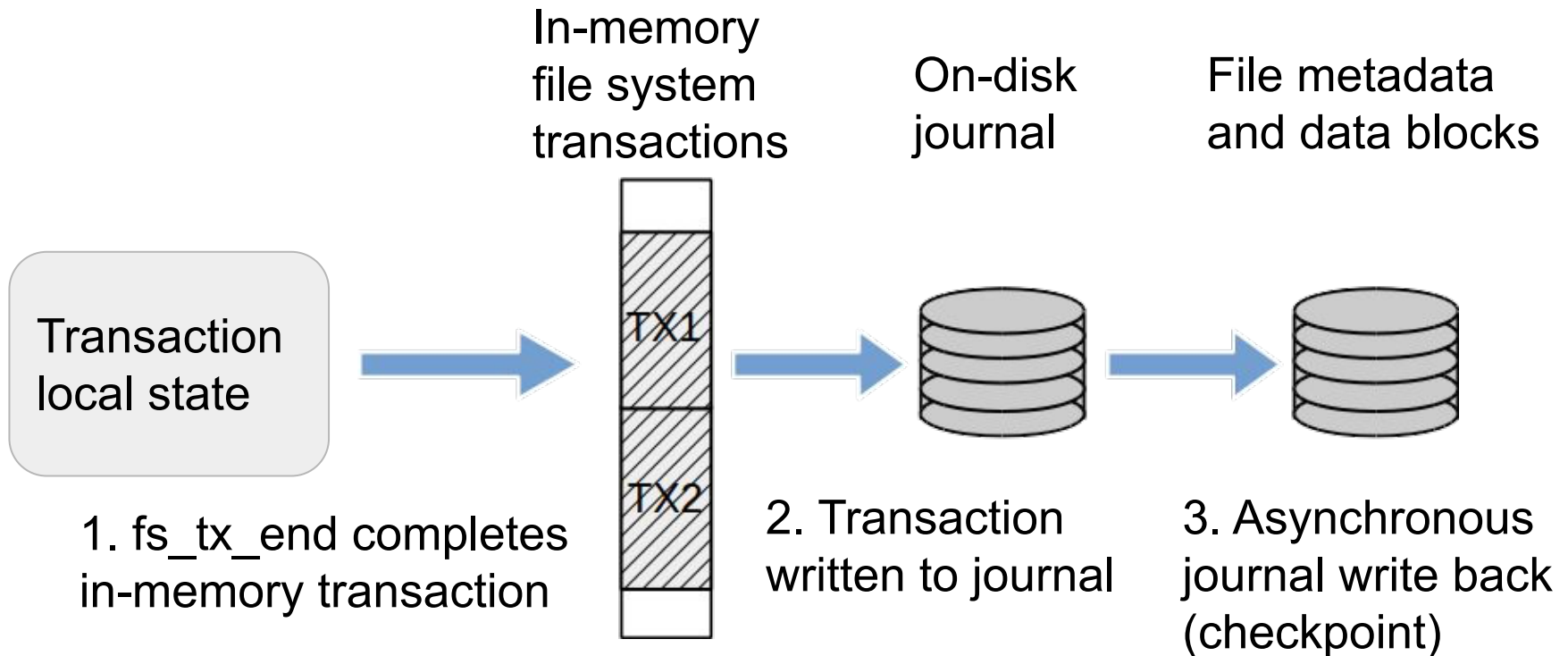
```
if (hFile == INVALID_HANDLE_VALUE)
{
    cerr << "CreateFileTransacted failed" << endl;
    return 1;
}
```

CloseHandle(hFile);

CommitTransaction(hTrans);
CloseHandle(hTrans);

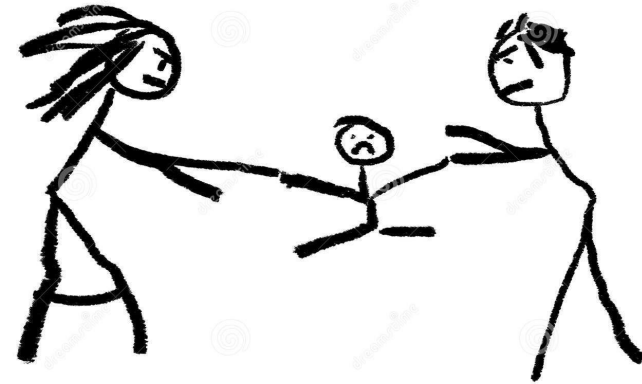
T2FS managing and persisting transactions

- Decreased complexity: use the file systems' crash consistency mechanism to create transactions.
 - Ext4 journal or ZFS copy-on-write



Isolation and Conflict detection

- In-progress writes are all local to kernel thread
- Eager conflict detection on inodes, directory entries
 - Enables flexible contention management
- Fine-grained page locks
 - More scalable than reader/writer lock



T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.

Raw files

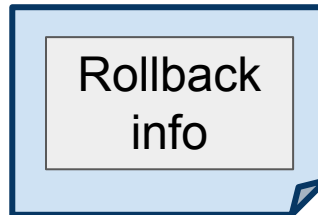
Attachment file



```
1.create(/dir/attachment)
  write(/dir/attachment)
  fsync(/dir/attachment)
  fsync(/dir/)
```

SQLite

Roll-back log



```
2.create(/dir/journal)
  write(/dir/journal)
  fsync(/dir/journal)
  fsync(/dir/)
  /*safe append*/
  fsync(/dir/journal)

4.unlink(/dir/journal)
```

Database file

	/dir/attachment

```
3.write(/dir/db)
  fsync(/dir/db)
```

T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.

Raw files

Attachment file



```
1.create(/dir/attachment)
  write(/dir/attachment)
  fsync(/dir/attachment)
  fsync(/dir/)
```

SQLite

Database file

	/dir/attachment

```
3.write(/dir/db)
  fsync(/dir/db)
```

T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.

Raw files

Attachment file



```
1.create(/dir/attachment)
   write(/dir/attachment)
   fsync(/dir/attachment)
   fsync(/dir/)
```

SQLite

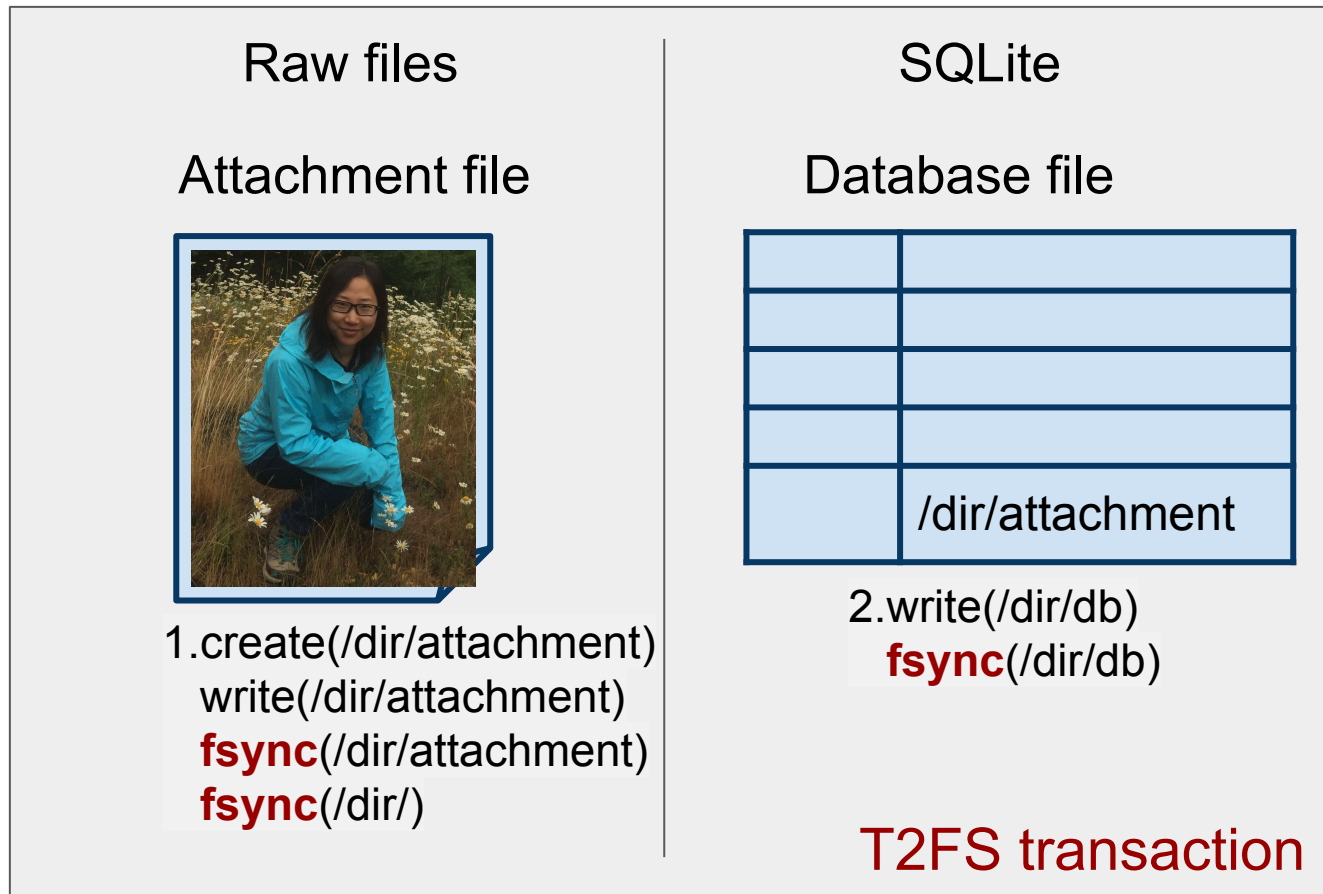
Database file

	/dir/attachment

```
2.write(/dir/db)
   fsync(/dir/db)
```

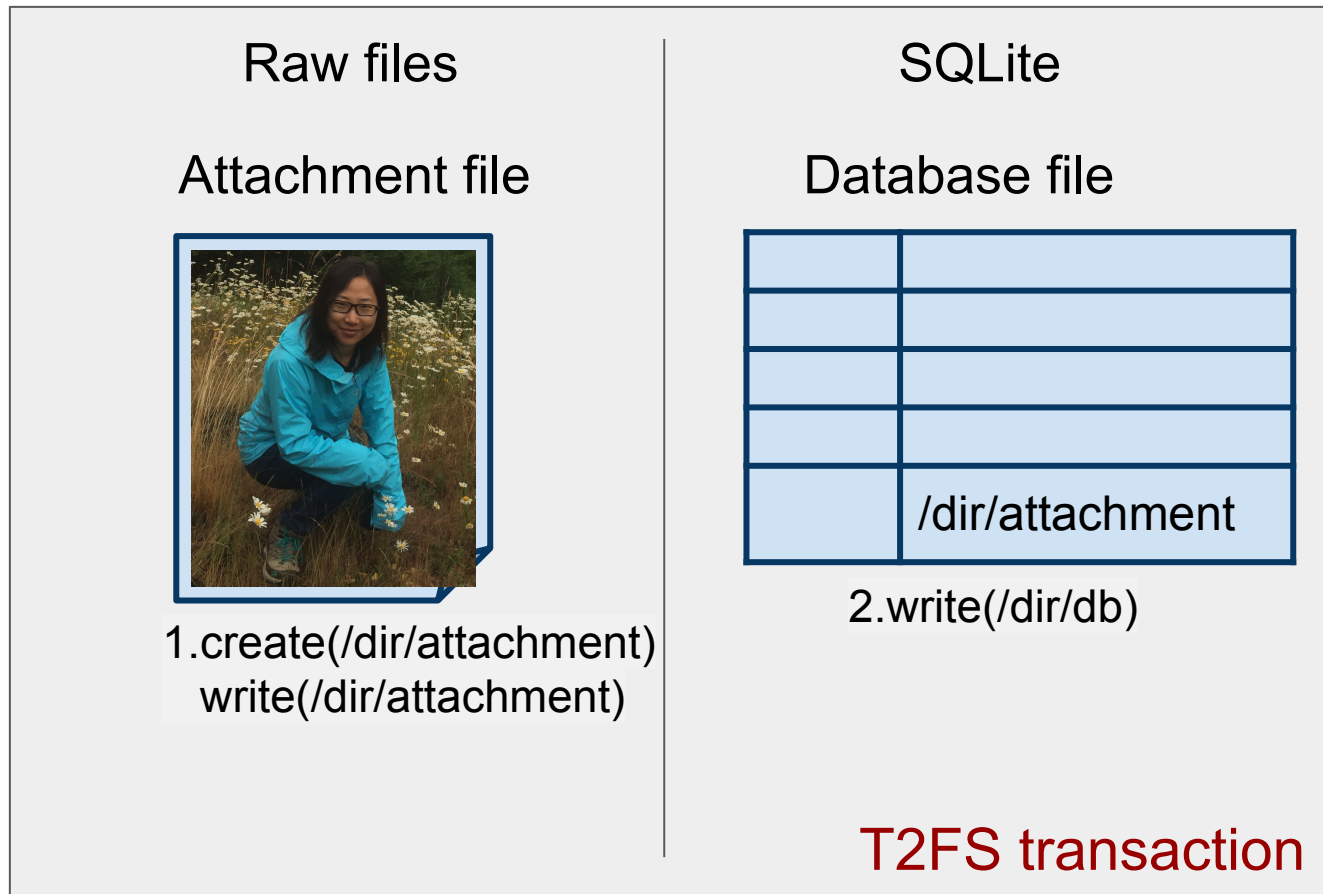
T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.



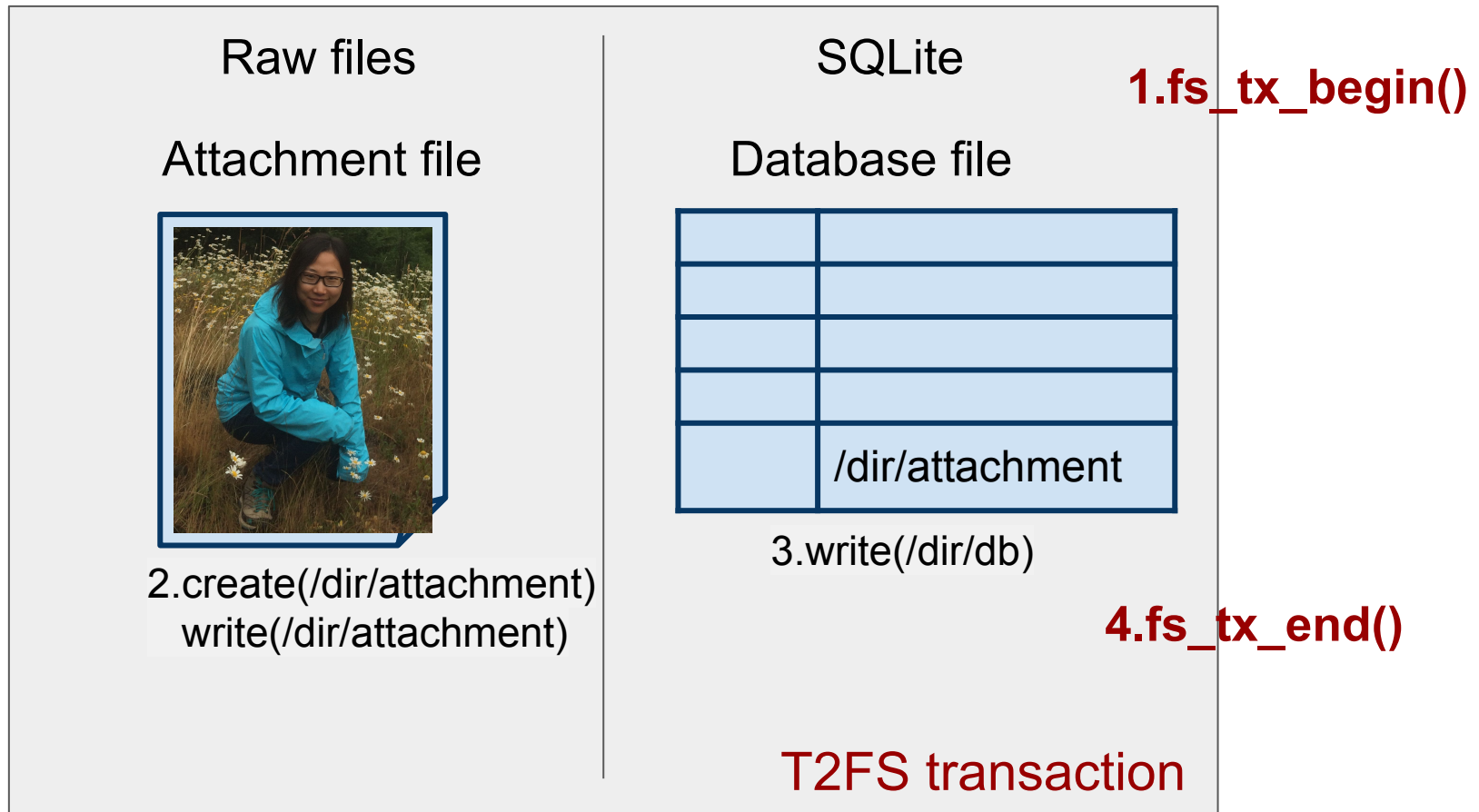
T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.



T2FS API: Cross-abstraction transactions

Modify the Android mail application to use T2FS transactions.



Evaluation: single-threaded SQLite



1.5M 1KB operations. 10K operations grouped in a transaction.
Database prepopulated with 15M rows.

Transactions as a foundation for other optimizations

- Enable automatic file system optimizations
 - Eliminate temporary durable files.
 - e.g. SQLite delete mode, directly wrapped by T2FS transaction
 - Consolidate IO across transactions.
 - Delay persistence during commit
- Use transactional mechanism to implement unrelated file system optimizations
 - Separate ordering from durability (osync [SOSP 13]).

Summary

Easy to use & deploy

Data safe on crash

ACID across abstractions

High performance

- Persistent data is structured; tough to make crash consistent
 - All data stored in the file system
- A transactional file system has the right API and mechanisms
- The file system journal makes implementing transactions easier
- Need transactions across storage abstractions

Thank you!