APPTUS

Invited lecture

Database tuning, ITU

2008-04-15

# Apptus Technologies

APPTUS

- Founded 2000 with HQ in Lund
- North America & UK offices
- 60 employees, revenue of 55 million SEK in 2007
- Specialized in search and database technology
- Focus on Yellow Pages services and E-commerce
- Clients include: Eniro, IKEA, European Directories, Yell, Quebecor, Yellow Pages Group, Lunarstorm, CDON, Bokus.com, Schibstedt, Handelsbanken, Astra Zeneca, DeGuleSider.dk

# Jesper Larsson

jesper.larsson@apptus.com

- 1969: born in Malmö
- 1990: started comp.sci./maths studies (Lund University)
- 1999: finished Ph.D. (suffix trees, data compression)
- 2000: Apptus Employee #7 (head of research)
- Worked with developing database/search platform ever since

# Outline

1. The relational data model & the model-implementation misconception
2. Relational text indexing in Apptus Theca®
3. Complexity of practical data structure implementation. Example: concurrency
4. The technology of Apptus Technologies: a historical survey. The DBMS deficiencies that have given us our business opportunities

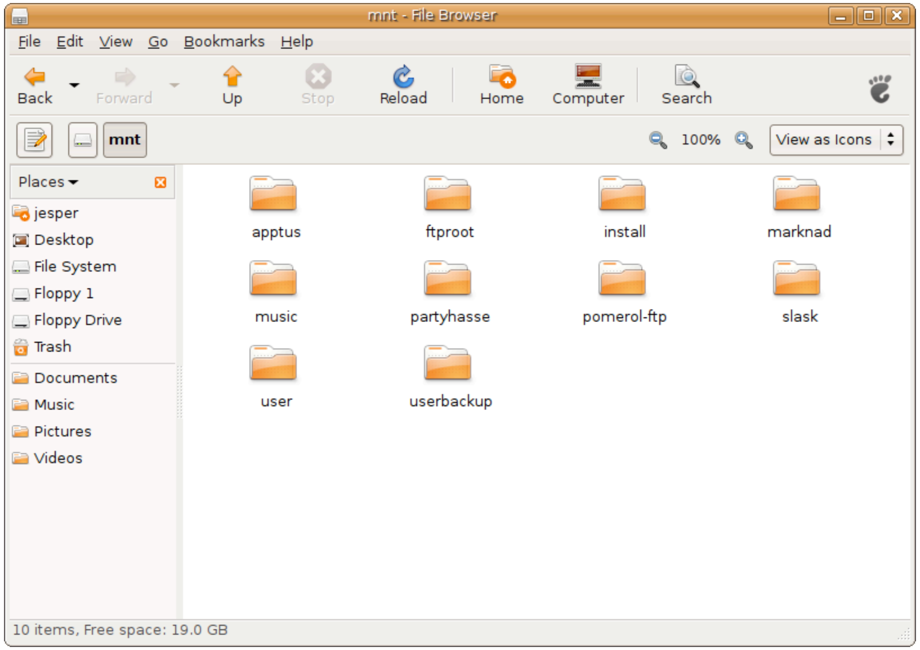
# What is a model?

- Simple abstract representation of something, often reusing concepts from other domains
- Framework for reasoning
- A good model helps us see clearly, draw conclusions
- A bad model obscures and confuses our thoughts

# The model is an interface

- Defines how something is used/exposed, NOT how it is implemented
- Use of *metaphors*

Example:



# Keep the model simple

- Easier to learn
- Easier to implement! Less redundant operations to code
- Easier to make efficient: put the effort where it is needed
- Enemies: “feature creeping”, “syntactic sugar”, big standards committees

# Relational data model

- Data correspond to *truth statements about individuals*
- Each statement corresponds to a combination of individuals – a *tuple*
- A relation is a set of tuples corresponding to true statements

# Misleading statement 1

“In the relational model, data are stored as tables”

- The model says nothing of storage
- Tables are just a way of drawing relations

Correct:

- In the relational model, data are perceived as relations, which may be *visualized* as tables.

# Misleading statement 2

SQL:

INSERT INTO persons VALUES ('Jesper', 'M', 'Malmö');

- Feels like *putting a thing into a container*…
- … rather than *declaring a truth*

Clearer (Prolog):

person(jesper, m, malmö).

# Text and relations

From overview lecture (#1):

- “the relational model is not well suited to represent the structure of text”

From text indexing lecture (#8):

- “Observations on inverted files: Can be implemented directly in the relational model”

- Both true?
- Any true?

# Text and relations

- One data representation:

```
"Observations on inverted files: Can be
  implemented directly in the relational
  model"
```

- Another:

```
{ (Observations, 1), (on, 2),
  (inverted, 3), (files, 4), (:, 5),
  (Can, 6), (be, 7), (implemented, 8),
  (directly, 9), (in, 10), (the, 11),
  (relational, 12), (model, 13) }
```

# Text and relations

| Id | Title |
|---|---|
| 1 | Art of war |
| 2 | War and peace |
| 3 | Modern art |

**Book titles**

| Word | Id |
|---|---|
| art | 1,3 |
| war | 1,2 |
| modern | 3 |
| peace | 2 |

**Inverted index**

| Word | Id |
|---|---|
| art | 1 |
| art | 3 |
| war | 1 |
| war | 2 |
| modern | 3 |
| peace | 2 |

**Relational form**

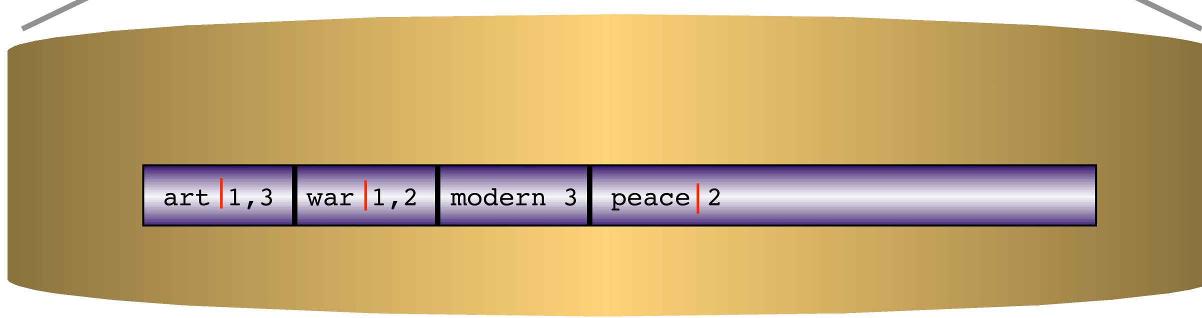
Relational model inefficient for storing text index?

# Separation of layers

**Logical representation**

| Word | Id |
|---|---|
| art | 1 |
| art | 3 |
| war | 1 |
| war | 2 |
| modern | 3 |
| peace | 2 |

**Physical representation**



# Text indexing in Theca

"Document" – anything that can produce "tokens"

→ tokenization – produce "tokens" out of "atomic" document

→ relation (document, token) used as input to the relational database

# Special text storage

- Supports subset of RDBMS operations
- Adds no non-relational “power”

# Example

```
rql> create table document using btree('id')
     as id posint, text string;
ok
rql> create table text_ix using freetext('word')
     derived as select id, word from document elab word(text);
Table text_ix will ignore delete
rql> commit;
ok
rql> insert into document values id, text {
     (1, 'Art of War'),
     (2, 'War and Peace'),
     (3, 'Modern Art') };
ok
rql> commit;
ok
```

# Example

```
rql> select * from text_ix;
values[(word, string), (id, posint)]{
 ('Art', 1),
 ('Art', 3),
 ('Modern', 3),
 ('Peace', 2),
 ('War', 1),
 ('War', 2),
 ('and', 2),
 ('of', 1)
}
```

# Example

```
rql> select id, text from document, text_ix where word='Art';
values[(id, posint), (text, string)]{
 (1, 'Art of War'),
 (3, 'Modern Art')
}
rql> select id, text from document,
     (select id from text_ix where word='Art'),
     (select id from text_ix where word='War');
values[(id, posint), (text, string)]{
 (1, 'Art of War')
}
```

# Storage class

```
rql> select class_name from _table_alias where
   alias='freetext';
values[(class_name, string)]{
 ('com.theca.adapters.RQLFreetext')
}
```

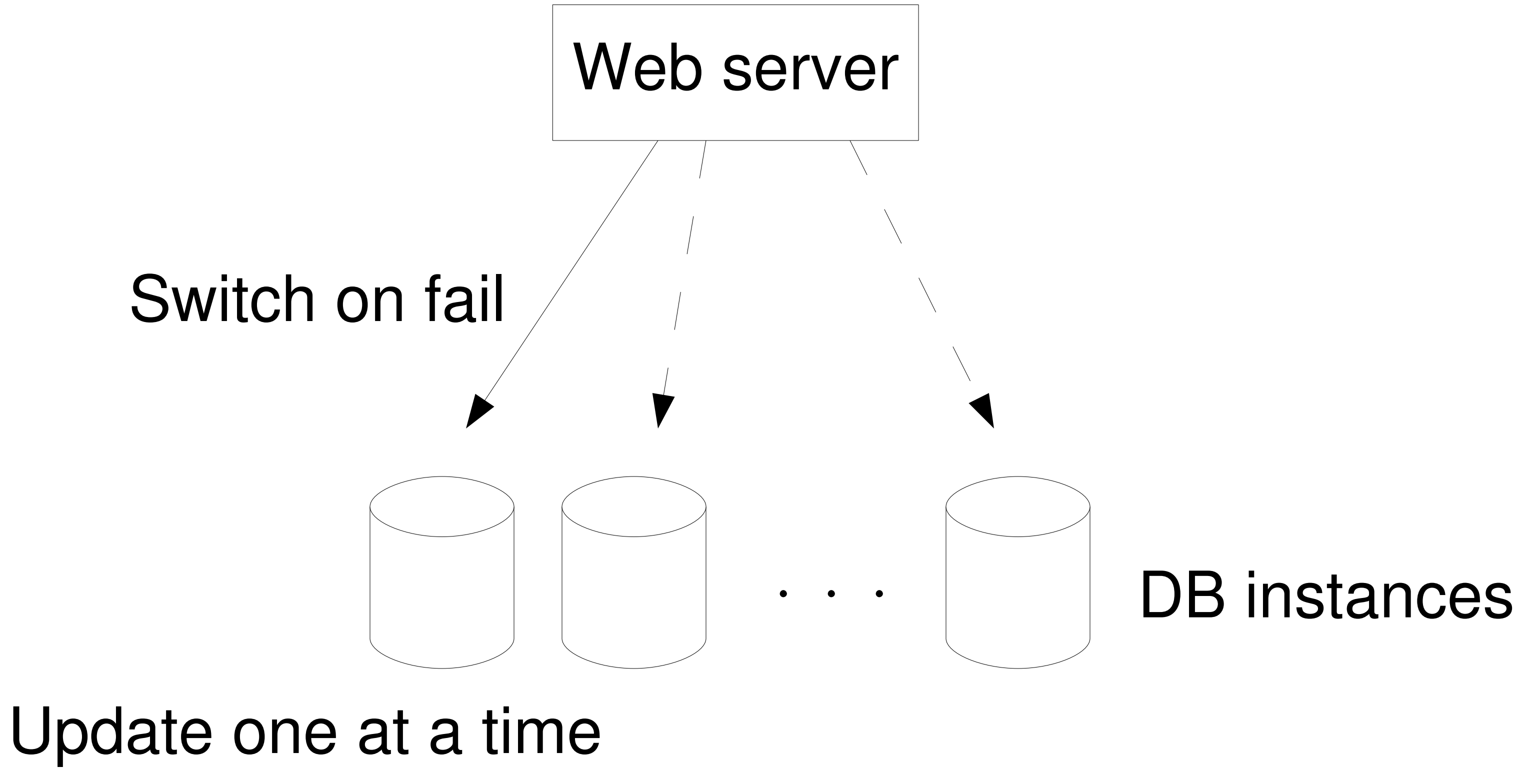
Class that implements “Storage” interface

# Concurrency and text search

APPTUS

- Theca (currently) uses block locking (e.g. B-tree node)
- System with high demands on response time (~100 ms) cannot wait for update transaction to finish

# Solution example 1



# Solution example 2

- Read uncommitted
- “Bad” hits in index get filtered out by join with document relation

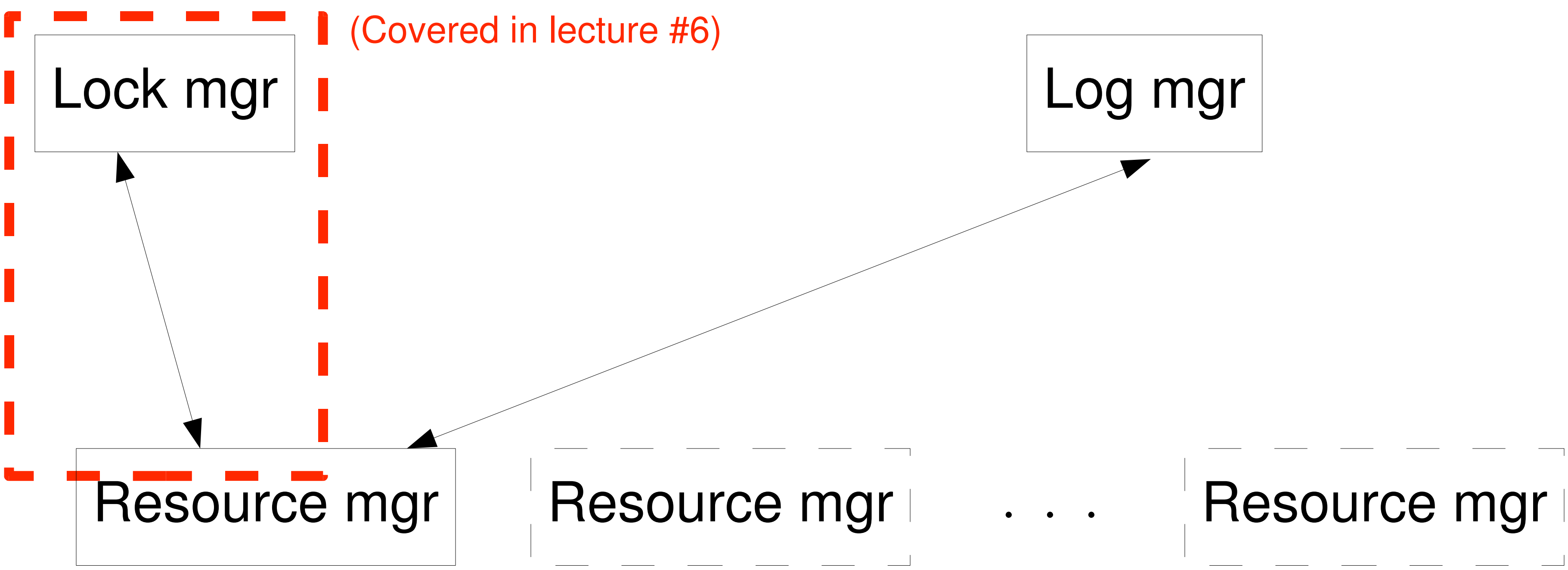
Note: both solutions imperfect, but that's (usually) ok

# Implementation challenges

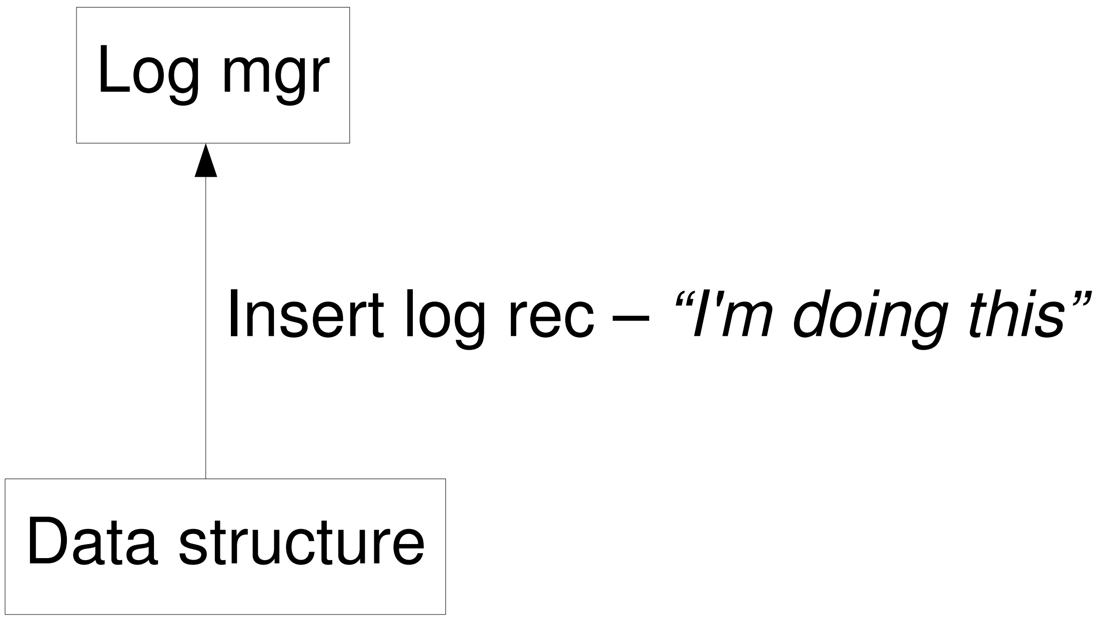
- “Book” data structures are simplified
- “We assume…”
- Variable-length keys/data
- Concurrency

# Concurrency

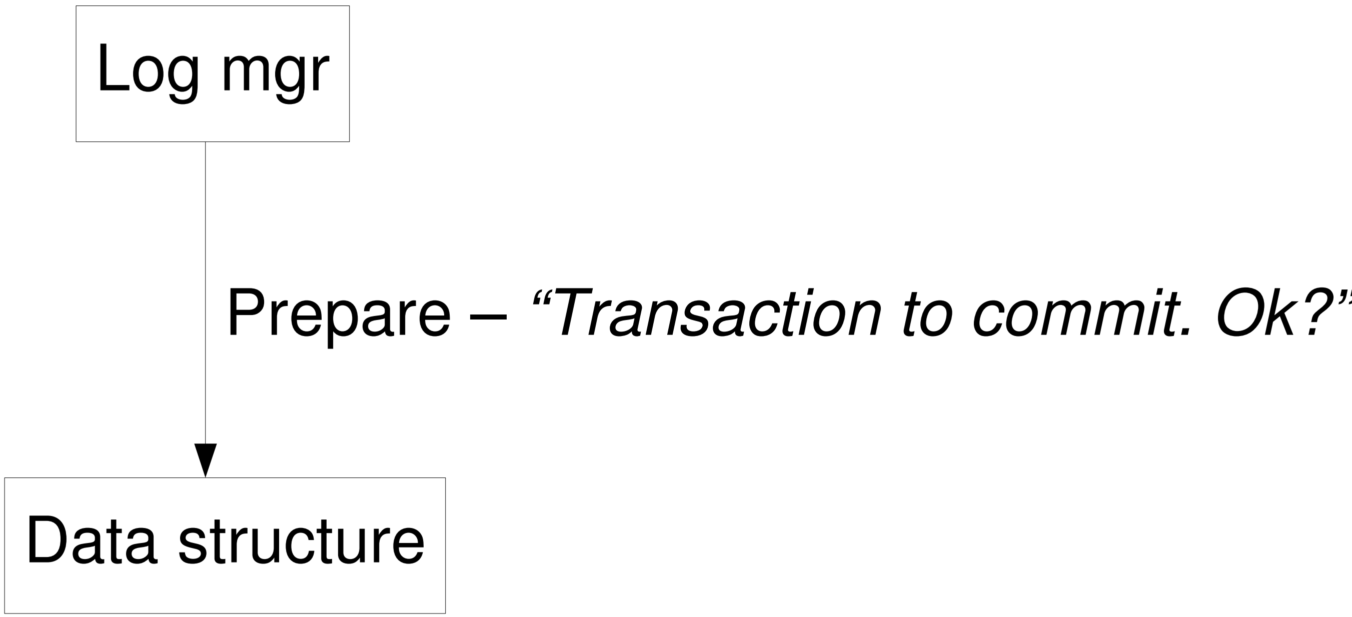
- *Resource manager*: part of data structure (or other resource) that interacts with transaction system



# Normal operation

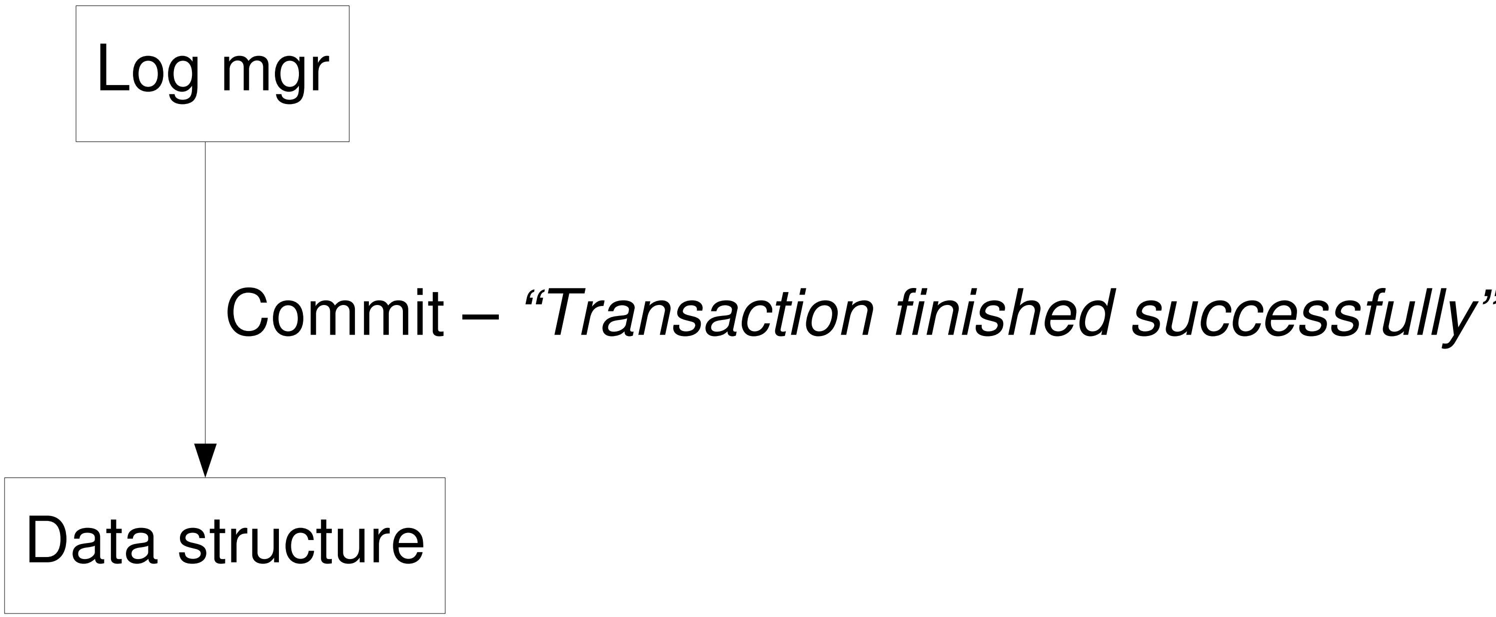


# Normal operation




If everyone ok, transaction commits, otherwise aborts

# Normal operation




Never asked to go back to earlier state

# Normal operation

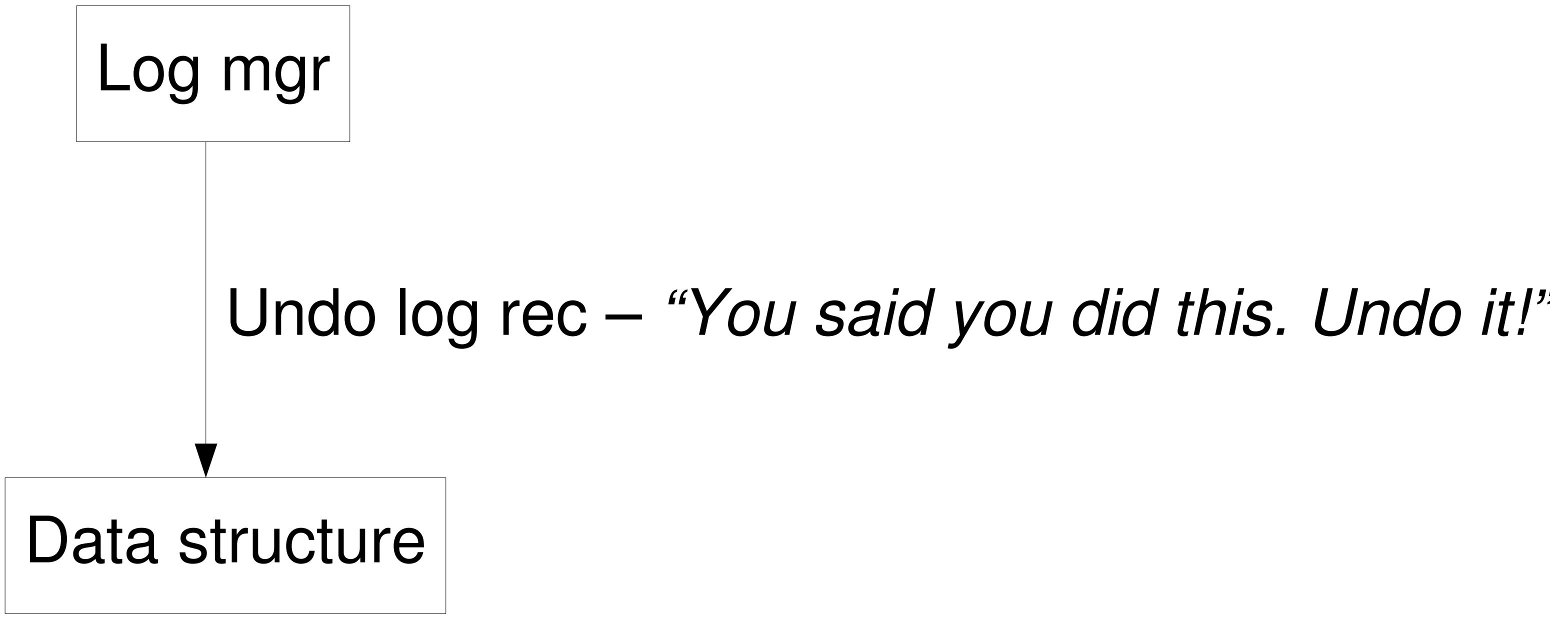
Log mgr

Checkpoint – *"Make sure you can recover from this point (flush to disk)"*

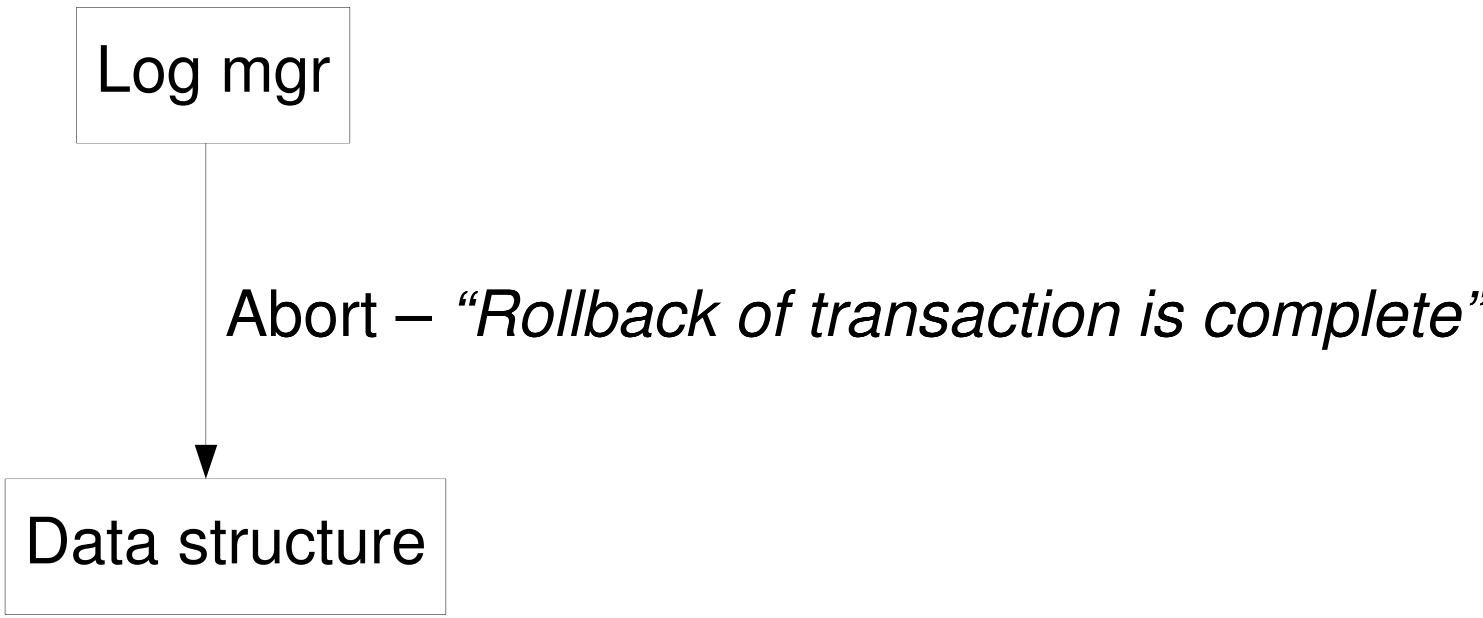
Data structure

Triggers flush in buffer mgr (also a resource mgr)

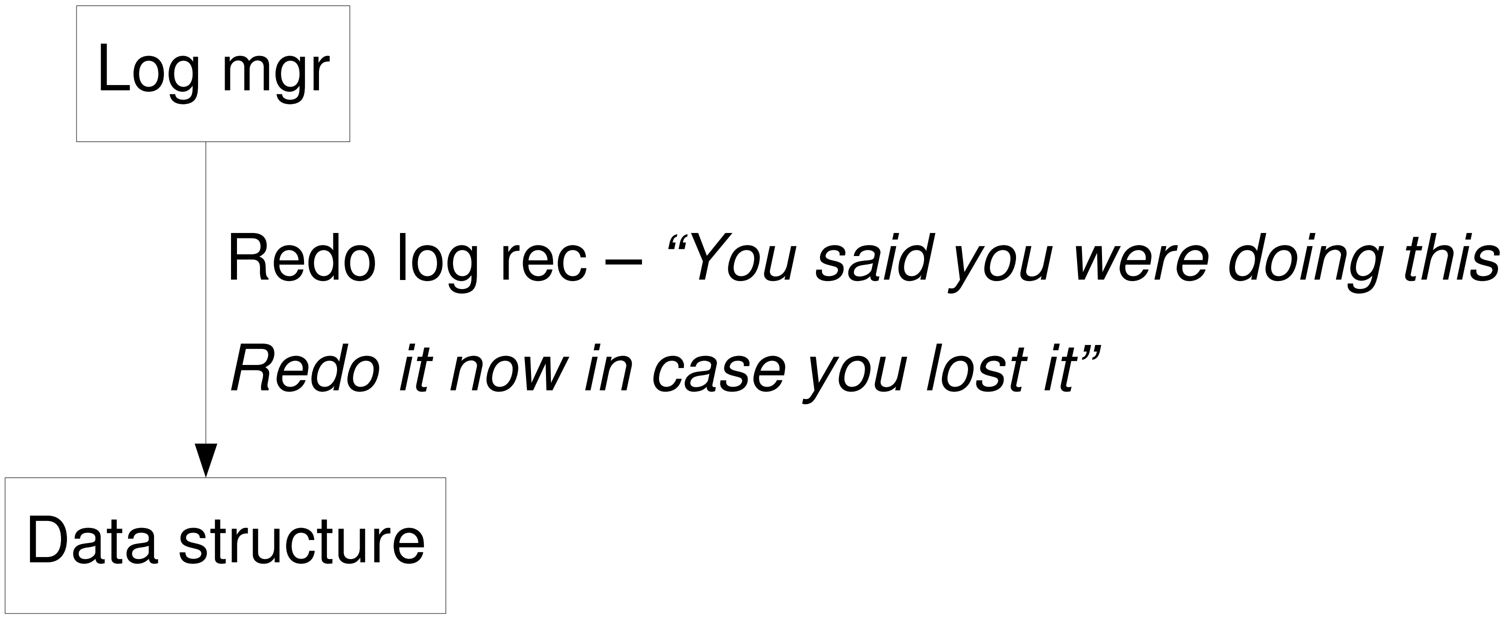
# Abort/rollback



# Abort/rollback



# Restart phase 1: REDO



# Restart phase 2: UNDO

Log mgr

Undo log rec – *"I just told you to redo this, but now I need you to undo it"*

Data structure

(For log records of uncommitted transactions)

# Resource manager interface

Data structure needs to implement:

```
void redo(log record)
void undo(log record)
boolean prepare()
void checkpoint(log record number)
```

informational:

```
void abort()
void commit()
```

# Log manager interface

```
void joinWork(resource manager id)
void logInsert(log record)
void logFlush()
```

# Logging is no big performance issue

- Bottleneck: log flush
- Needed on:
  - Start transaction
  - Commit
  - Data write to disk
- Data structure does not (usually) write directly to disk
  - Modifies buffer in memory
  - Buffer manager flushes buffer only when needed

# Update scenarios

- Small non-concurrent updates
  - Nothing to worry about!
- Large updates
  - Log write time ~ block access time
  - Logging is not bottleneck

# Update scenarios

- Small concurrent updates
  - Trick: “boxcarring”
  - Idea: mostly write full log blocks – performance like streaming data to disk
  - Implementation:

```
logInsert(log rec) {
   add log rec to buffer
   if (buffer full) {
      write to disk
      wake up waiting threads
   }
}
```

```
logFlush() {
   wait at most 1 ms
   if (no write during wait) {
      write to disk
      wake up waiting threads
   }
}
```

⇒ Overhead during low load, but that's no problem

# Dealing with logging bugs

- Very difficult to test
- Bugs lead to corrupt data
- Bugs may show up after months or years

# So what to do?

Banks etc. must not face lost data. Strategy:

- Keep full backups of safe states
- Log everything done (since backup) on high level

# Consequence of complexity

- Implementing new data structures may be painful
- You end up rather reusing already tested code

# Why are general DBMS so slow?

- Difficult question; no principal reason why they must be

A couple of answers:

- Slow for a task unless specifically optimized for it
- Declarative language (e.g. SQL) not used for the full query
  - Sometimes impossible, sometimes awkward
  - Optimizer does not see the full query!