Week 09

Assignment 2

Assignment 2

Implement a signature-based filtering scheme

- using superimposed codeword signatures
- three types: tuple-level, page-level, bit-sliced

We give you the overall framework, you supply the details

Once working, experimental analysis of signature performance

- create several database instances
- run benchmark PMR queries for each signature scheme
- measure costs (sigs read, pages read, tuple comparisons, false matches)

... Assignment 2

What's in the framework ...

- applications: create, insert, query, gendata, stats
- ADTs: hash, bits, tsig, psig, bsig, page, tuple, reln, query

We deal with individual relations, where each relation ...

- is comprised of unique tuples, all the same size, e.g.

  ```
  1000079,QHogGVRvjQRPMXbhKKfg,a3-25,a4-16
  ```

- with schema (id:int/unique, name:char(20), a_3:char7, ...)
- and where attributes 3..n all have the same structure
- stored as char strings, but with no trailing `'\0'`

... Assignment 2

Relations have a number of parameters:

- page size (fixed, defined by `PAGESIZE`)
- tuple size (fixed, determined by #attributes *n*) ⇒ *c*
- #tuples = *r* (dynamic, determined by #inserts)
- #pages = *b* (dynamic, determined by *r* and *c*)
- signature size = *m* (fixed, determined by *n* and p_F)
- bit-slice size (dynamic**, determined by *b*)

** but not in this assignment (fixed at 4K bits ⇒ no more than 4k pages)

... Assignment 2

Formulae for determining signature sizes:

- #bits / attribute = k = $1/log_e 2 \,.\, log_e (1/p_F)$
- #bits in tuple sig = m_t = $(1/log_e 2)^2 \,.\, n \,.\, log_e (1/p_F)$
- #bits in page sig = m_p = $(1/log_e 2)^2 \,.\, n \,.\, c \,.\, log_e (1/p_F)$
- #bits in bit-slice = b ** (except that we fix it to 4K bits)

Implementation: round sig/slice sizes up to multiple of 8 (bits/byte)

i.e. if m_t = 12, allocate 16 bits (2 bytes) for signature

Above values are computed when relation is created, and stored in params file

---

... Assignment 2

6/55

./gendata #Tuples #Attrs [StartID] [Seed]

- generates *#Tuples*, each with *#Attrs* attributes
- default starting ID is 1000000; can change with *StartID*
- can change seed for random # generator (default is 0)
- write tuples to standard output as comma-separated fields

Example:

```
$ ./gendata  5  4
1000000,lrfkQyuQFjKXyQVNRTyS,a3-00,a4-00
1000001,FrzrmzlYGFvEulQfpDBH,a3-01,a4-01
1000002,lqDqrrCRwDnXeuOQqekl,a3-02,a4-02
1000003,AITGDPHCSPIjtHbsFyfv,a3-03,a4-03
1000004,lADzPBfudkKlrwqAOzMi,a3-04,a4-04
$
```

... Assignment 2

7/55

./create RelName #Attrs 1/pF

- creates a relation called *RelName*
- initially with zero tuples; grows via `insert`
- all tuples added at the end of the last page
- all pages are full, except the last; no deletions

./stats RelName

- displays information about relation *RelName*

./dump RelName

- displays parameters; can be used to recreate *RelName*
- displays tuples from database, one per line

... Assignment 2

8/55

./insert RelName

- reads tuples, on per line, from standard input

- insert each tuple into *RelName*
- all tuples added at the end of the last page
- if last page is full, add a new page at end
- all pages are full, except the last; no deletions

... Assignment 2

`./query RelName PMR-query SigType`

- displays all tuples that match *PMR-query*
- queries have the form $x_1, x_1, ... x_n$
  - where each x_i is either a value or `?`
- queries can return 0 or more tuples

Example queries, assuming *n=3*

- `?,?,?` ... matches all tuples
- `1000000,?,?` ... matches tuples with $attr_1$ = 1000000
- `?,abcde,?` ... matches tuples with $attr_2$ = "abcde"
- `?,abcde,a3-01` ... matches tuples with $attr_2$ = "abcde" and $attr_2$ = "a3-01"

... Assignment 2

`./query RelName PMR-query SigType`

- displays matching tuples, one-per-line
- *SigType* determines what kind of signatures are used
  - *SigType* = `t` ... use tuple-level signatures
  - *SigType* = `p` ... use page-level signatures
  - *SigType* = `b` ... use bit-sliced signatures
- after displaying result tuples, should also display:
  - number of signature/bit-slice pages read
  - number of data pages read
  - number of tuples checked for matching
  - number of data pages read but with no matching tuples

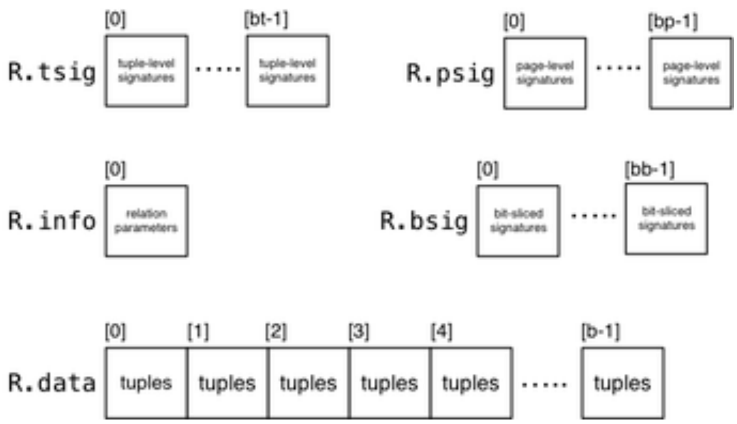
... Assignment 2

Relations are implemented as five files:

- *Rel*`.info` ... one page with relation parameters
- *Rel*`.data` ... pages containing tuples
- *Rel*`.tsig` ... pages containing tuple signatures
- *Rel*`.psig` ... pages containing page signatures
- *Rel*`.bsig` ... pages containing bit slices

... Assignment 2

File structures:



... Assignment 2

13/55

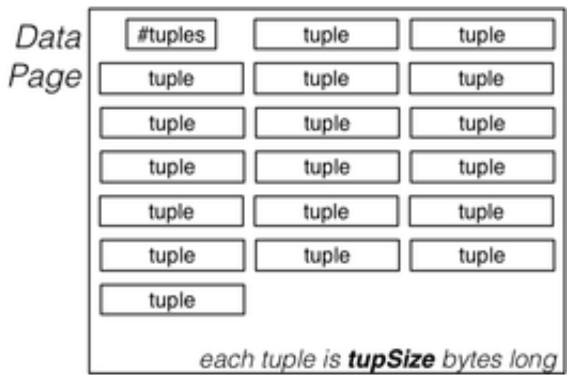
Contents of relation `.info` file

```
Count  npages;   // number of data pages (dynamic)
Count  ntups;    // total number of tuples (dynamic)
Count  nattrs;   // number of attributes (fixed)
Count  tupSize;  // # bytes in tuples (all same size)
Count  tupPP;    // max tuples per page
Count  tk;       // bits set per attribute
Count  tm;       // width of tuple signature (bits)
Count  tsigSize; // # bytes in tuple signature
Count  tsigPP;   // max tuple signatures per page
Count  pm;       // width of page signature (bits)
Count  psigSize; // # bytes in page signature
Count  psigPP;   // max tuple signatures per page
Count  bm;       // width of bit-slice
Count  bsigSize; // # bytes in bit-slice
Count  bsigPP;   // max bit-slices per page
```

... Assignment 2

14/55

Page contents (data file):

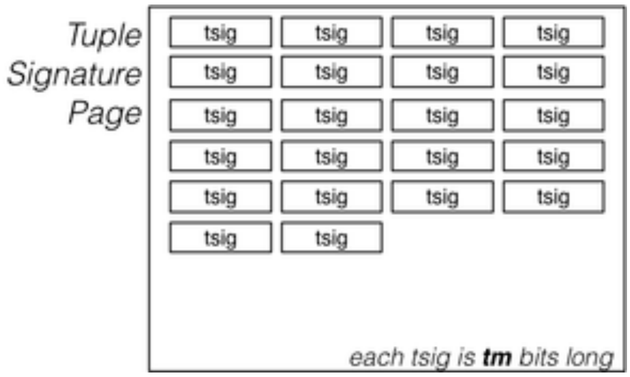


There are up to `ntups` tuples in each page

... Assignment 2

15/55

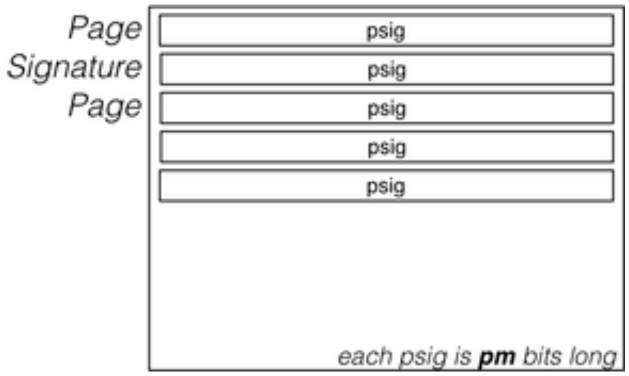
Page contents (tuple-level signature file):



There are up to `tsigPP` signatures per page

... Assignment 2

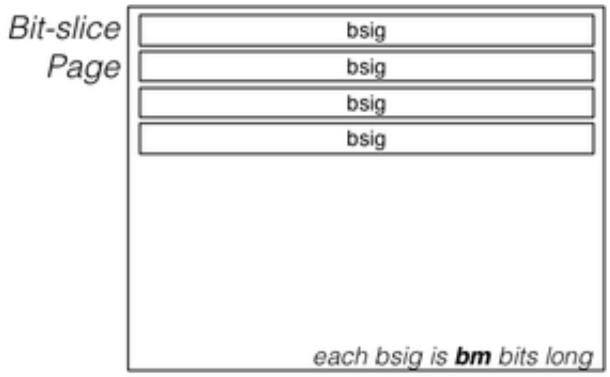
Page contents (page-level signature file):



There are up to `psigPP` signatures per page

... Assignment 2

Page contents (bit-sliced signature file):



in theory, bm == npages

... Assignment 2

Query-time data structure (minimal):

```
struct QueryRep {
    // static info
```

```
    Reln    rel;        // need to remember Relation info
    char    *qstring;   // query string
    //dynamic info
    Bits    pages;      // list of pages to examine
    PageID  curpage;    // current page in scan
    Count   curtup;     // current tuple within page
    // statistics info
    ... you can put here whatever you need
    ... to produce the required statistics
};
```

This is effectively the iteration structure described previously.

... Assignment 2

19/55

`hash.c`

- hash function from PostgreSQL
- produces a 32 bit integer given a string

`util.c`

- definition of `fatal()` error handler

`defs.h`

- global definitions (e.g. `PAGESIZE`, `Count`)

... Assignment 2

20/55

Abstract Data Types ... interface provided, you implement

- **`bits.h`** ... operations on long bit-strings (`Bits`)
- **`tsig.h`** ... operations on tuple signatures
- **`psig.h`** ... operations on page signatures
- **`bsig.h`** ... operations on bit-slices
- **`page.h`** ... operations on pages (`Page`)
- **`tuple.h`** ... operations on tuples (`Tuple`)
- **`reln.h`** ... operations on relations (`Reln`)
- **`query.h`** ... operations for queries (`Query`)

... Assignment 2

21/55

Style of implementing ADTs

- interface defines type as a pointer to a `struct`
- implementation defines `struct` details

Example:

```
In Bits.h

typedef struct _BitsRep *Bits;

In Bits.c

struct _BitsRep {
```

```
    Count  nbits;          // how many bits
    Count  nbytes;         // how many bytes in array
    Byte   bitstring[1];   // array of bytes to hold bits
                           // actual array size is nbytes
};
```

Transactions: the story so far

Transactions should obey ACID properties

Isolation can be compromised by uncontrolled concurrency

Serializable schedules avoid potential anomalies

- less safe (more concurrent) isolation levels exist
- read uncommitted, read committed, repeatable read

Styles of concurrency control

- locking (two-phase, deadlock)
- optimistic concurrency control (try, then fix problems)
- multi-version concurrency control (less locking needed)

Implementing Atomicity/Durability

Atomicity/Durability

Reminder:

Transactions are *atomic*

- if a tx commits, all of its changes persist in DB
- if a tx aborts, none of its changes occur in DB

Transaction effects are *durable*

- if a tx commits, its effects persist
  (even in the event of subsequent (catastrophic) system failures)

Implementation of atomicity/durability is intertwined.

Durability

What kinds of "system failures" do we need to deal with?

- single-bit inversion during transfer mem-to-disk
- decay of storage medium on disk (some data changed)
- failure of entire disk device (data no longer accessible)
- failure of DBMS processes (e.g. `postgres` crashes)
- operating system crash; power failure to computer room
- complete destruction of computer system running DBMS

The last requires off-site *backup*; all others should be locally recoverable.

... Durability

26/55

Consider following scenario:

Desired behaviour after system restart:

- all effects of T1, T2 persist
- as if T3, T4 were aborted (no effects remain)

... Durability

27/55

Durabilty begins with a *stable disk storage subsystem*

- i.e. effects of `putPage()` and `getPage()` are consistent

We can prevent/minimise loss/corruption of data due to:

- mem/disk transfer corruption: parity checking
- sector failure: mark "bad" blocks
- disk failure: RAID (levels 4,5,6)
- destruction of computer system: off-site backups

Dealing with Transactions

28/55

The remaining "failure modes" that we need to consider:

- failure of DBMS processes or operating system
- failure of transactions (`ABORT`)

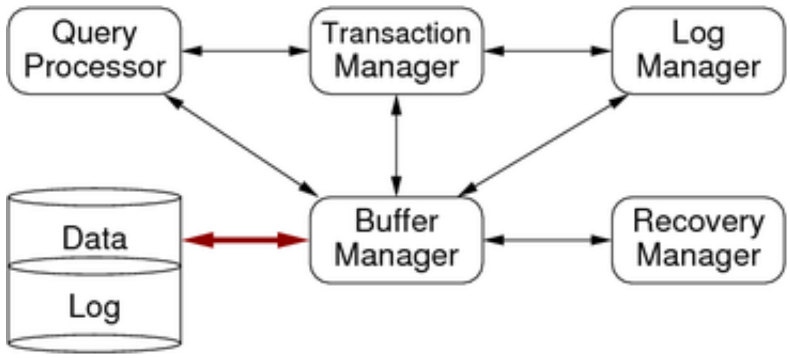
Standard technique for managing these:

- keep a *log* of changes made to database
- use this log to restore state in case of failures

Architecture for Atomicity/Durability

29/55

How does a DBMS provide for atomicity/durability?



Execution of Transactions

30/55

Transactions deal with three address spaces:

- stored data on the disk (representing DB state)
- data in memory buffers (where held for sharing)
- data in their own local variables (where manipulated)

Each of these may hold a different "version" of a DB object.

PostgreSQL processes make heavy use of shared buffer pool

⇒ transactions do not deal with much local data.

... Execution of Transactions

31/55

Operations available for data transfer:

- `INPUT(X)` ... read page containing `X` into a buffer
- `READ(X,v)` ... copy value of `X` from buffer to local var `v`
- `WRITE(X,v)` ... copy value of local var `v` to `X` in buffer
- `OUTPUT(X)` ... write buffer containing `X` to disk

`READ/WRITE` are issued by transaction.

`INPUT/OUTPUT` are issued by buffer manager (and log manager).

`INPUT/OUTPUT` correspond to `getPage()/putPage()` mentioned above

... Execution of Transactions

32/55

Example of transaction execution:

```
-- implements A = A*2; B = B+1;
BEGIN
READ(A,v); v = v*2; WRITE(A,v);
READ(B,v); v = v+1; WRITE(B,v);
COMMIT
```

`READ` accesses the buffer manager and may cause `INPUT`.

`COMMIT` needs to ensure that buffer contents go to disk.

... Execution of Transactions

States as the transaction executes:

```
t   Action        v  Buf(A)  Buf(B)  Disk(A)  Disk(B)
-----------------------------------------------------
(0) BEGIN         .     .       .        8        5
(1) READ(A,v)     8     8       .        8        5
(2) v = v*2      16     8       .        8        5
(3) WRITE(A,v)   16    16       .        8        5
(4) READ(B,v)     5    16       5        8        5
(5) v = v+1       6    16       5        8        5
(6) WRITE(B,v)    6    16       6        8        5
(7) OUTPUT(A)     6    16       6       16        5
(8) OUTPUT(B)     6    16       6       16        6
```

After tx completes, we must have either
Disk(A)=8, Disk(B)=5 or Disk(A)=16, Disk(B)=6

If system crashes before (8), may need to undo disk changes.
If system crashes after (8), may need to redo disk changes.

Transactions and Buffer Pool

Two issues arise w.r.t. buffers:

- *forcing* ... `OUTPUT` buffer on each `WRITE`
  - ensures durability; disk always consistent with buffer pool
  - poor performance; defeats purpose of having buffer pool
- *stealing* ... replace buffers of uncommitted tx's
  - if we don't, poor throughput (tx's blocked on buffers)
  - if we do, seems to cause atomicity problems?

Ideally, we want stealing and not forcing.

... Transactions and Buffer Pool

Handling *stealing*:

- transaction T loads page P and makes changes
- T_2 needs a buffer, and P is the "victim"
- P is output to disk (it's dirty) and replaced
- if T aborts, some of its changes are already "committed"
- must log values changed in P at "steal-time"
- use these to UNDO changes in case of failure of T

... Transactions and Buffer Pool

Handling *no forcing*:

- transaction T makes changes & commits, then system crashes
- but what if modified page P has not yet been output?
- must log changed values in P as soon as they change
- use these to support REDO to restore changes

Above scenario may be a problem, even if we are forcing

- e.g. system crashes immediately after requesting a `WRITE()`

Logging

37/55

Three "styles" of logging

- *undo* ... removes changes by any uncommitted tx's
- *redo* ... repeats changes by any committed tx's
- *undo/redo* ... combines aspects of both

All approaches require:

- a sequential file of log records
- each log record describes a change to a data item
- log records are written first
- actual changes to data are written later

Known as *write-ahead logging* (PostgreSQL uses WAL)

Undo Logging

38/55

Simple form of logging which ensures atomicity.

Log file consists of a *sequence* of small records:

- `<START T>` ... transaction `T` begins
- `<COMMIT T>` ... transaction `T` completes successfully
- `<ABORT T>` ... transaction `T` fails (no changes)
- `<T,X,v>` ... transaction `T` changed value of `X` from `v`

Notes:

- we refer to `<T,X,v>` generically as `<UPDATE>` log records
- update log entry created for each `WRITE` (not `OUTPUT`)
- update log entry contains *old* value (new value is not recorded)

... Undo Logging

39/55

Data must be written to disk in the following order:

1. `<START>` transaction log record
2. `<UPDATE>` log records indicating changes
3. the changed data elements themselves
4. `<COMMIT>` log record

Note: sufficient to have `<T,X,v>` output before `X`, for each `X`

... Undo Logging

40/55

For the example transaction, we would get:

```
t    Action        v  B(A)  B(B)  D(A)  D(B)  Log
--------------------------------------------------------
(0)  BEGIN         .    .     .     8     5   <START T>
(1)  READ(A,v)     8    8     .     8     5
```

```
(2)  v = v*2      16    8     .     8     5
(3)  WRITE(A,v)   16   16     .     8     5    <T,A,8>
(4)  READ(B,v)     5   16     5     8     5
(5)  v = v+1       6   16     5     8     5
(6)  WRITE(B,v)    6   16     6     8     5    <T,B,5>
(7)  FlushLog
(8)  StartCommit
(9)  OUTPUT(A)     6   16     6    16     5
(10) OUTPUT(B)     6   16     6    16     6
(11) EndCommit                                 <COMMIT T>
(12) FlushLog
```

Note that T is not regarded as committed until (12) completes.

... Undo Logging

41/55

Simplified view of recovery using UNDO logging:

- scan *backwards* through log
    - if `<COMMIT T>`, mark `T` as committed
    - if `<T,X,v>` and `T` not committed, set `X` to `v` on disk
    - if `<START T>` and `T` not committed, put `<ABORT T>` in log

Assumes we scan entire log; use checkpoints to limit scan.

... Undo Logging

42/55

Algorithmic view of recovery using UNDO logging:

```
committedTrans = abortedTrans = startedTrans = {}
for each log record from most recent to oldest {
    switch (log record) {
    <COMMIT T> : add T to committedTrans
    <ABORT T>  : add T to abortedTrans
    <START T>  : add T to startedTrans
    <T,X,v>    : if (T in committedTrans)
                     // don't undo committed changes
                 else  // roll-back changes
                     { WRITE(X,v); OUTPUT(X) }
}   }
for each T in startedTrans {
    if (T in committedTrans) ignore
    else if (T in abortedTrans) ignore
    else write <ABORT T> to log
}
flush log
```

Checkpointing

43/55

Simple view of recovery implies reading entire log file.

Since log file grows without bound, this is infeasible.

Eventually we can delete "old" section of log.

- i.e. where *all* prior transactions have committed

This point is called a *checkpoint*.

- all of log prior to checkpoint can be ignored for recovery

... Checkpointing

44/55

Problem: many concurrent/overlapping transactions.

How to know that all have finished?

1. periodically, write log record `<CHKPT (T1,..,Tk)>`
   (contains references to all active transactions ⇒ active tx table)
2. continue normal processing (e.g. new tx's can start)
3. when all of `T1,..,Tk` have completed,
   write log record `<ENDCHKPT>` and flush log

Note: tx manager maintains chkpt and active tx information

... Checkpointing

45/55

Recovery: scan backwards through log file processing as before.

Determining where to stop depends on ...

- whether we meet `<ENDCHKPT>` or `<CHKPT...>` first

If we encounter `<ENDCHKPT>` first:

- we know that all incomplete tx's come after prev `<CHKPT...>`
- thus, can stop backward scan when we reach `<CHKPT...>`

If we encounter `<CHKPT (T1,..,Tk)>` first:

- crash occurred *during* the checkpoint period
- any of `T1,..,Tk` that committed before crash are ok
- for uncommitted tx's, need to continue backward scan

Redo Logging

46/55

Problem with UNDO logging:

- all changed data must be output to disk before committing
- conflicts with optimal use of the buffer pool

Alternative approach is *redo* logging:

- allow changes to remain only in buffers after commit
- write records to indicate what changes are "pending"
- after a crash, can apply changes during recovery

... Redo Logging

47/55

Requirement for redo logging: *write-ahead rule*.

Data must be written to disk as follows:

1. start transaction log record
2. update log records indicating changes

3. then commit log record (`OUTPUT`)
4. then `OUTPUT` changed data elements themselves

Note that update log records now contain `<T,X,v'>`,
where `v'` is the *new* value for `X`.

... Redo Logging

48/55

For the example transaction, we would get:

```
t     Action         v  B(A)  B(B)  D(A)  D(B)  Log
--------------------------------------------------------
(0)   BEGIN          .    .     .     8     5   <START T>
(1)   READ(A,v)      8    8     .     8     5
(2)   v = v*2       16    8     .     8     5
(3)   WRITE(A,v)    16   16     .     8     5   <T,A,16>
(4)   READ(B,v)      5   16     5     8     5
(5)   v = v+1        6   16     5     8     5
(6)   WRITE(B,v)     6   16     6     8     5   <T,B,6>
(7)   COMMIT                                    <COMMIT T>
(8)   FlushLog
(9)   OUTPUT(A)      6   16     6    16     5
(10)  OUTPUT(B)      6   16     6    16     6
```

Note that T is regarded as committed as soon as (8) completes.

... Redo Logging

49/55

Simplified view of recovery using REDO logging:

- identify all committed tx's (backwards scan)
- scan *forwards* through log
  - if `<T,X,v>` and `T` is committed, set `X` to `v` on disk
  - if `<START T>` and `T` not committed, put `<ABORT T>` in log

Assumes we scan entire log; use checkpoints to limit scan.

Undo/Redo Logging

50/55

UNDO logging and REDO logging are incompatible in

- order of outputting `<COMMIT T>` and changed data
- how data in buffers is handled during checkpoints

Undo/Redo logging combines aspects of both

- requires new kind of update log record
  `<T,X,v,v'>` gives both old and new values for `X`
- removes incompatibilities between output orders

As for previous cases, requires write-ahead of log records.

Undo/redo loging is common in practice; Aries algorithm.

... Undo/Redo Logging

51/55

For the example transaction, we might get:

```
t    Action        v  B(A)  B(B)  D(A)  D(B)  Log
--------------------------------------------------------
(0)  BEGIN         .    .     .     8     5   <START T>
(1)  READ(A,v)     8    8     .     8     5
(2)  v = v*2      16    8     .     8     5
(3)  WRITE(A,v)   16   16     .     8     5   <T,A,8,16>
(4)  READ(B,v)     5   16     5     8     5
(5)  v = v+1       6   16     5     8     5
(6)  WRITE(B,v)    6   16     6     8     5   <T,B,5,6>
(7)  FlushLog
(8)  StartCommit
(9)  OUTPUT(A)     6   16     6    16     5
(10)                                          <COMMIT T>
(11) OUTPUT(B)     6   16     6    16     6
```

Note that T is regarded as committed as soon as (10) completes.

... Undo/Redo Logging

52/55

Simplified view of recovery using UNDO/REDO logging:

- scan log to determine committed/uncommitted txs
- for each uncommitted tx `T` add `<ABORT T>` to log
- scan *backwards* through log
  - if `<T,X,v,w>` and `T` is not committed, set `X` to `v` on disk
- scan *forwards* through log
  - if `<T,X,v,w>` and `T` is committed, set `X` to `w` on disk

... Undo/Redo Logging

53/55

The above description simplifies details of undo/redo logging.

Aries is a complete algorithm for undo/redo logging.

Differences to what we have described:

- log records contain a sequence numnber (LSN)
- LSNs used in tx and buffer managers, and stored in data pages
- additional log record to mark `<END>` (of commit or abort)
- `<CHKPT>` contains only a timestamp
- `<ENDCHKPT..>` contains tx and dirty page info

Recovery in PostgreSQL

54/55

PostgreSQL uses write-ahead undo/redo style logging.

It also uses multi-version concurrency control, which

- tags each record with a tx and update timestamp

MVCC simplifies some aspects of undo/redo, e.g.

- some info required by logging is already held in each tuple
- no need to undo effects of aborted tx's; use old version

... Recovery in PostgreSQL

55/55

Transaction/logging code is distributed throughout backend.

Core transaction code is in **`src/backend/access/transam`**.

Transaction/logging data is written to files in **`PGDATA/pg_xlog`**

- a number of very large files containing log records
- old files are removed once all txs noted there are completed
- new files added when existing files reach their capacity (16MB)
- number of tx log files varies depending on tx activity

Produced: 20 Sep 2018