Distributed Database Recovery

Physical Models

CSC 321/621 – 4/12/2012

Failure Transparency

- Distributed database must ensure transactions are atomic and durable, even when facing new failure possibilities of:
  - Loss of a network message
  - Failure of a network connection
  - Failure of a site
- We will assume that the underlying network protocols resolve issue 1 for us

2PC: Stages

- Voting stage:
  - Coordinator asks all participants if ready to commit
  - If anyone votes “ABORT” or never responds (up to some time period), everyone will be asked to abort
  - If all vote “COMMIT”, everyone will be asked to commit

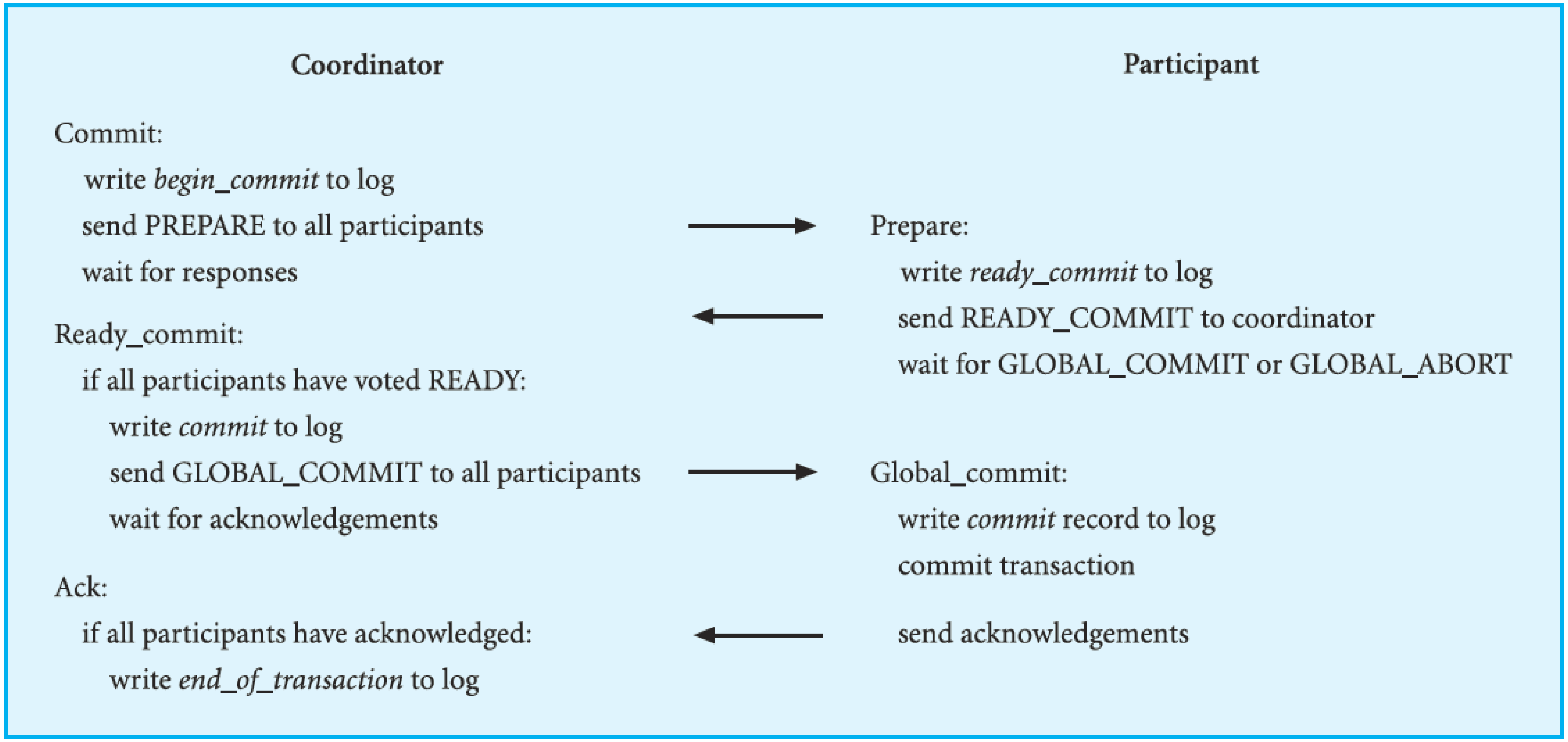
2PC: Stages

- Decision Stage:
  - Assume a vote has been made
  - Anyone that has voted to ABORT can actually precede on their own to abort immediately (as they will eventually be issued a request to abort)
  - Otherwise, wait for decision from coordinator

  - Decision is either GLOBAL COMMIT or GLOBAL ABORT
  - Each local site can use its own log to rollback or commit as necessary

2PC: Stages (Interaction Diagram & Details)

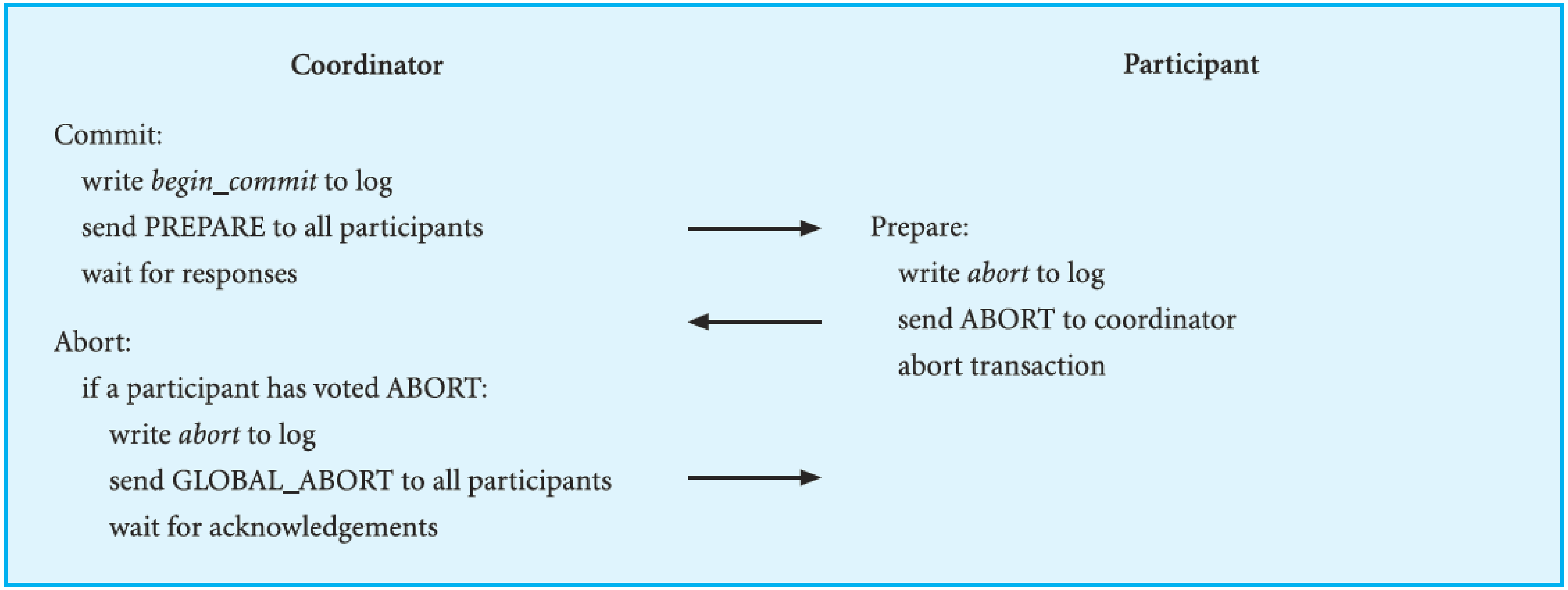
- The coordinator and participant interaction for a participant voting COMMIT



Coordinator		Participant
Commit:		
write *begin_commit* to log		
send PREPARE to all participants	→	Prepare:
wait for responses		write *ready_commit* to log
	←	send READY_COMMIT to coordinator
Ready_commit:		wait for GLOBAL_COMMIT or GLOBAL_ABORT
if all participants have voted READY:		
write *commit* to log		
send GLOBAL_COMMIT to all participants	→	Global_commit:
wait for acknowledgements		write *commit* record to log
		commit transaction
Ack:		
if all participants have acknowledged:	←	send acknowledgements
write *end_of_transaction* to log		

2PC: Stages (Action Diagram &Details)

- The coordinator and participant interaction for a participant voting ABORT



```
                Coordinator                                            Participant

Commit:
   write begin_commit to log
   send PREPARE to all participants            ———→      Prepare:
   wait for responses                                       write abort to log
                                                ←———        send ABORT to coordinator
Abort:                                                      abort transaction
   if a participant has voted ABORT:
      write abort to log
      send GLOBAL_ABORT to all participants    ———→
      wait for acknowledgements
```

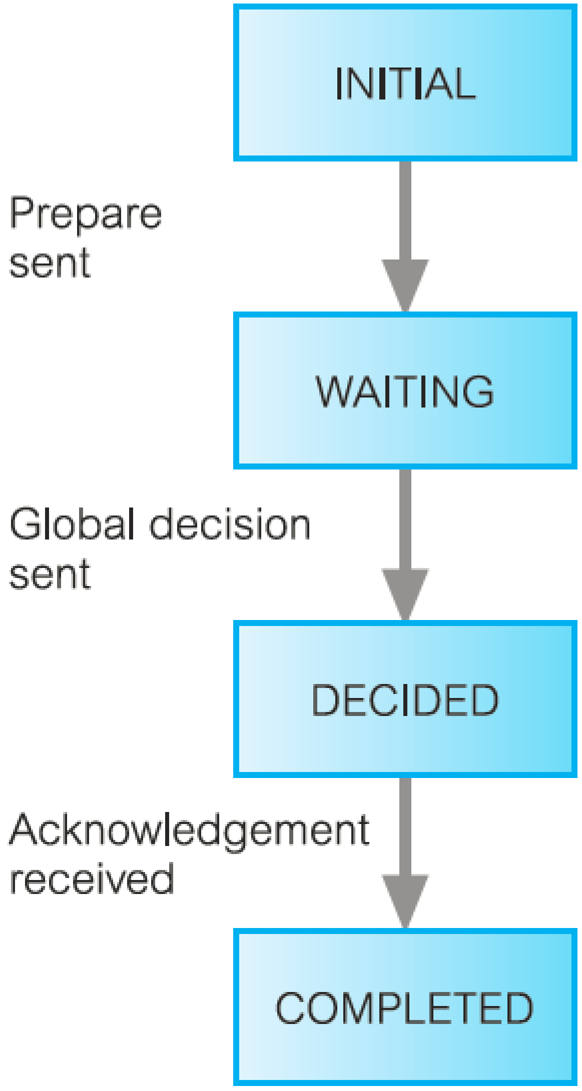
2PC: Stages

- Note extensive use of log writes to local logs
  - Record keeping of progress through process

- There are lots of places where problems could occur while the voting & decision making is going on
  - What if the coordinator fails?
  - What if a local site fails?
- Deal with these from POV of still-operational machines

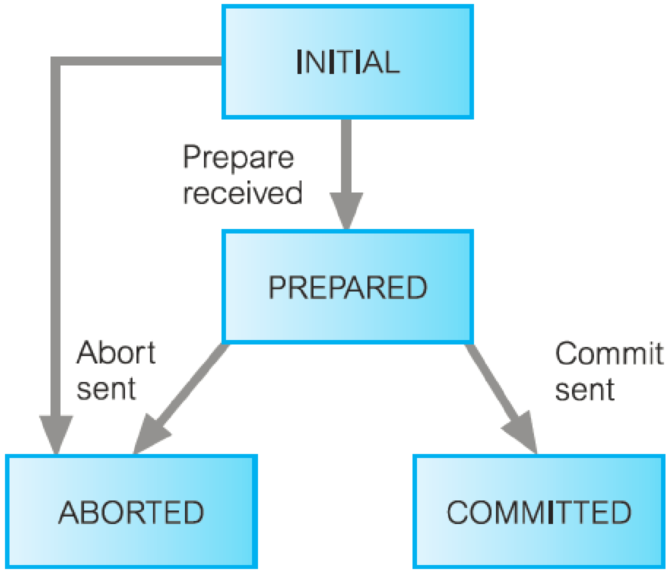
2PC: Stages & Termination Protocols

- Coordinator can be in one of four states:
  - INITIAL, WAITING, DECIDED, COMPLETED
- From the Coordinator POV, with respect to local site going down:
  - (WAITING) If any site fails to vote on what to do with the transaction within a certain time window, the coordinator triggers a GLOBAL ABORT.
  - (DECIDED) If a site fails to acknowledge what it did with the transaction within a certain time window, the global decision is resent until all acknowledgements are received.



2PC: Stages & Termination Protocols

- Participant can be in one of four states:
  - INITIAL, PREPARED, ABORTED COMMITTED
- From the Local Site POV, with respect to coordinator going down:
  - (INITIAL) The site times out waiting for the "please vote" [prepare] instruction for this transaction – do a unilateral abort and if the actual "please vote" instruction comes, vote ABORT
  - (PREPARED) The site times out waiting for the global decision
    - If local decision was to abort, not a problem (can run it/already did run it)
    - If local decision was to commit, can't change it's mind and not commit but can't go ahead and commit since don't know global decision, so either BLOCK (do nothing until hear something later) or ask others if they know the decision.



2PC: Recovery Protocols

- Previous slides dealt with “still-operational machines”; how do the failed machines recover and get to a consistent state?

- Again deal with:
  - Coordinator failure
  - Participant failure

2PC: Recovery Protocols - Coordinator

- Coordinator can be in one of four states:
  - INITIAL, WAITING, DECIDED, COMPLETED
- If coordinator failed in:
  - Initial state: Nothing has even happened yet, so just begin the 2PC procedure
  - Waiting state: Assuming it has not heard an abort response before it crashed, restart the 2PC procedure
    - Re-asking isn't problematic (though being slow about it, may get some aborts back)
  - Decided state:
    - If had already received back all acknowledgements, just complete transaction
    - If not, resend global decision to non-acknowledged sites

2PC: Recovery Protocols - Participant

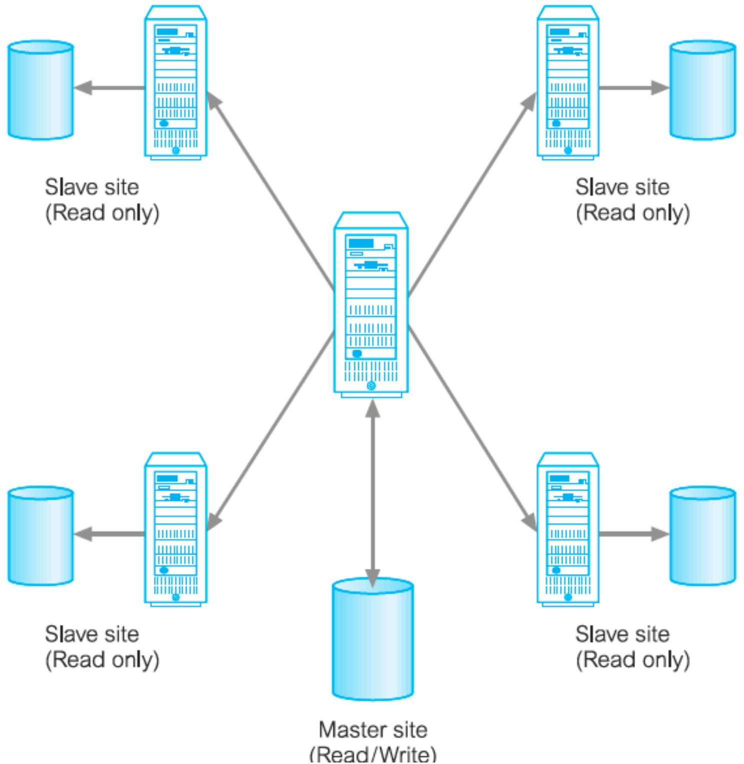
- Participant can be in one of four states:
  - INITIAL, PREPARED, ABORTED, COMMITTED
- If participant failed in:
  - Initial state: Not yet voted, so unilaterally abort, will cause others to abort
  - Prepared state: Has already voted, so needs to follow through with GLOBAL vote results (which will be sent/resent at some point)
  - Aborted or committed state: Has already fully processed the transaction locally (committed or aborted), so nothing to do.

Replication

- Previous slides basically hinted at notion of "sychronous replication"
  - Any updates to data in which there were replicate copies must update the replicates as well
    - Writes to N sites instead of 1 site
    - Discussed locking of all copies
    - 2-phase commit: replicate copies are part of the process
  - Necessary in some domains (financial information) but expensive in terms of communication and highly susceptible to site failures when heavily replicated
- Asynchronous replication:
  - Only requires that replicate copies are eventually updated

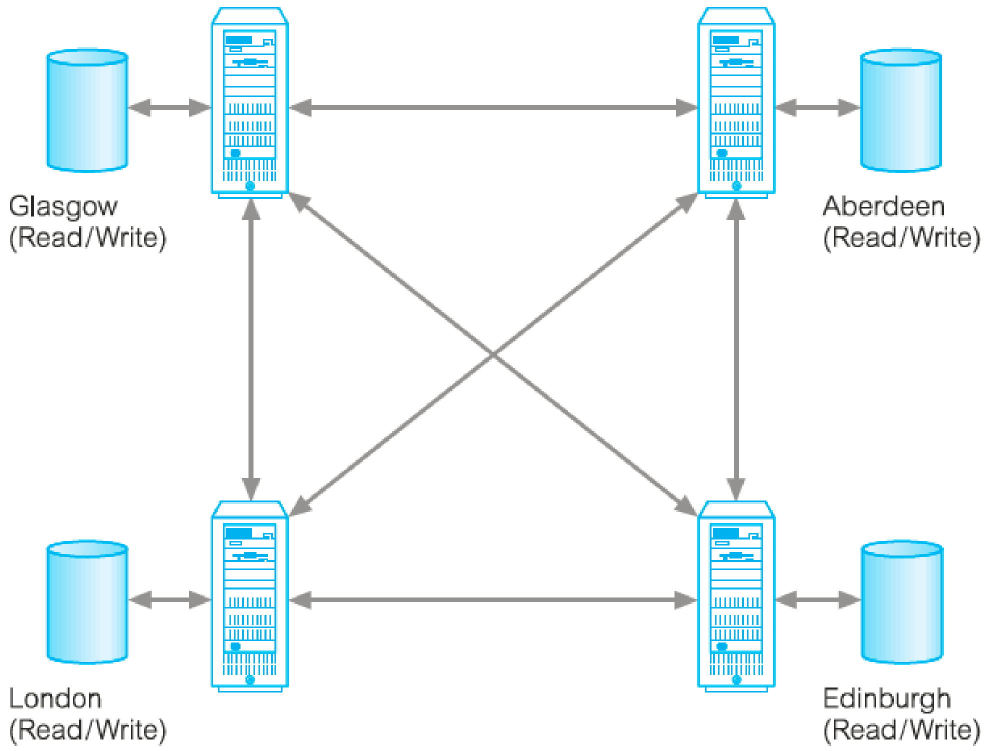
Ensuring Replication: A Few Models

- When making changes to a database: Master/slave, publisher/subscriber --
  - Writes to a replicated chunk made to a single site, read only copies propagated to other sites



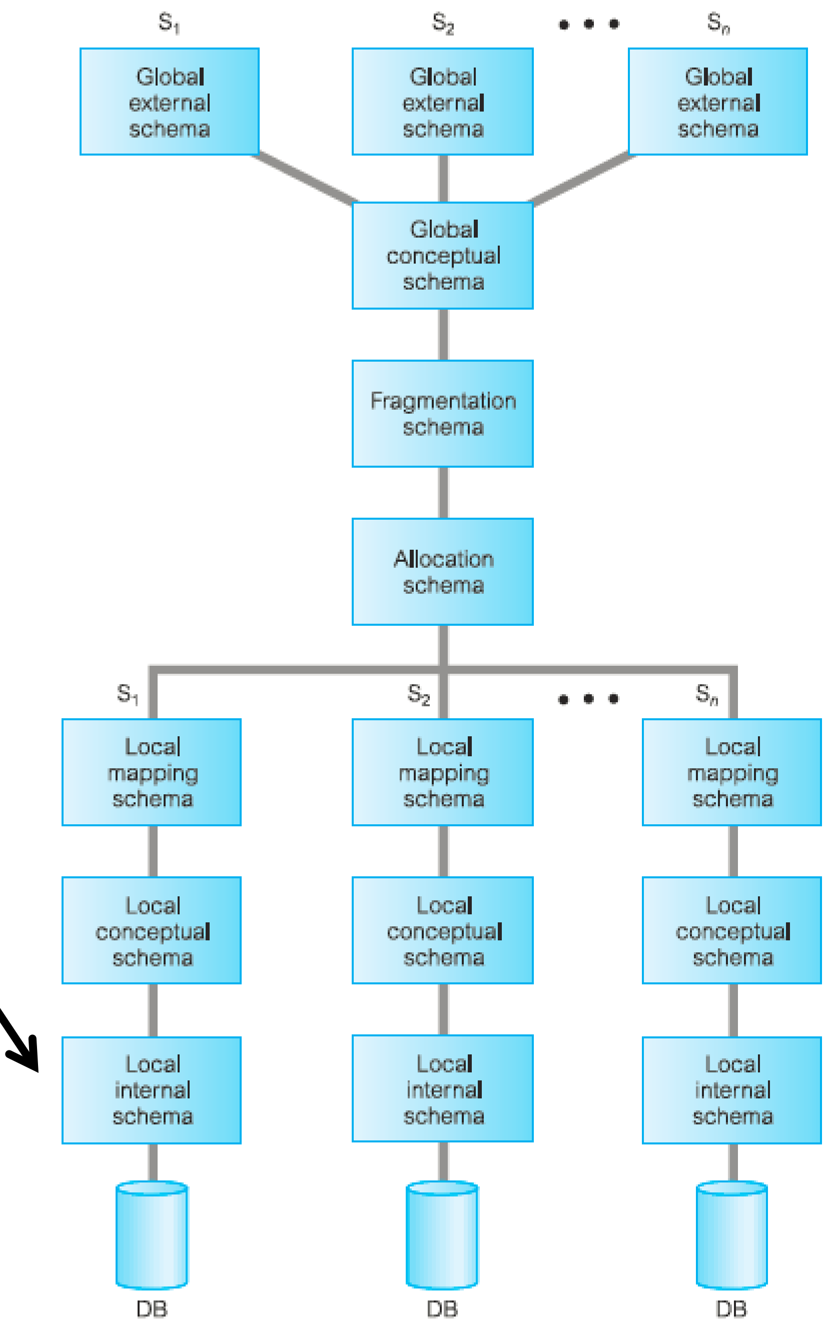
Ensuring Replication: A Few Models

- When making changes to a database: Update-anywhere
  - Writes can be made at any site, with data propagated around
  - P2P effectively



Physical Models of Database

What's going on "behind the scenes"?



Physical Models: Some Terminology

- Stored field: Smallest unit of stored data – essentially a (row,column) entry in table.
  - Instances of part numbers in a Parts table
- Stored record: A collection of related stored fields – essentially a tuple
  - A full description of a given part in a Parts table
- Stored file: A collection of all occurrences of one type of stored record – essentially, a table

Physical Models: Some Terminology

- Think in OO terminology:
  - Field ➔ member variable
  - Record ➔ an object/struct (actually this term was frequently used to mean struct), which is a combination of fields
  - File ➔ a file holding a collection of instances of some record type (all “Automobile” data together in one file)

Physical Models: More Terminology

- Page: A fixed size disk chunk
- Page set: A collection of pages

- A page set may hold one or more stored files
  - A table (stored file) could actually cover more than one page

- These are actual physical disk chunks!

Common Page and File Actions

- Page:
  - Retrieve a page from a page set
  - Replace a page within a page set
  - Add a new page to a page set (moving it from a free list)
  - Remove a page from a page set (moving it to a free list)
- File:
  - Retrieve a record from a file
  - Replace a record in a file
  - Add a new record to a file
  - Remove a record from a file
  - Create a new file
  - Destroy a file

Same basic fundamental operations as “containers”,
databases proper (insert, update, delete, query…)   *They are all just stores of information*

Heap (Unordered) File

- A heap file is a file which is ordered only by insertion time
  - Unordered by any attribute of the data
- Insertion:
  - Add on last page of file, making new page if needed
- Deletion:
  - Leave a gap in a file/page where removed (may go back and compress later)
- Query:
  - Search linearly through the set of records for match

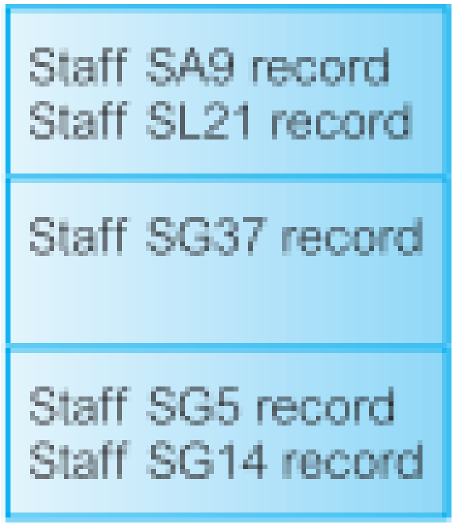
- Poor choice, except for bulk loading of data (as inserts are typically fast)

Hashing

- Employ a hash function to compute where stored record (tuple) should be placed
  - One example: divide numerical representation of primary key by a #, place item at remainder value location
  - %13 ➔ 13 potential spots
- Basically, same notion as hashtables from CSC 221
- Inserts/retrieves/removes all employ the hash function to (ideally) have O(1) access to the data

Hashing

- Physical sequence of records will likely have not relation to any sequencing on attributes
- Very likely will have gaps in files/pages
- O(1) is theoretical access time
- If a collision occurs (two records have same hash address), have to search for record
- Searching can be expensive, especially if new disk pages have to be loaded (remember, disk drive access is most expensive piece)
- Does anyone remember some of the collision management protocols for hashtables?



Collision Management

- Linear scan
  - Search downwards through table, one spot at a time; loop to top if hit bottom
  - If hit gap, item not already present
- Overflow pages
  - Have dumping ground pages that specifically hold all records that were involved in collisions (except for the ones that made it in w/o collision)
- Overflow chaining
  - Every spot in hashtable has a “pointer” (synonym pointer) which references a slot in a page which holds the first item that collided with the spot
  - Every slot pointed to has its own pointer, which points to a slot in a page which holds another piece of data that collided with the data of interest

Hashing: Reviewing Effectiveness

- Linear scan from hashed address
    - This may require you to look at items that don’t even have the same hash address as what you are looking for

- Overflow pages
    - Have pages that specifically hold all records that were involved in collisions (except for the ones that made it in w/o collision)
    - Linear search just through those records is better than linear search through whole table, but still may look at records that can’t possibly be a match (as these are records from any collision)

Hashing

- Overflow chaining:
  - Every spot in hashtable has a “pointer” (synonym pointer) which references a slot in a page which holds the first item that collided with the spot
  - Every slot pointed to has it’s own pointer, which points to a slot in a page which holds another piece of data that collided with the data of interest
  - Searching only through the data that has the same hash address – fewer searches!
    - Paying for it with extra memory for pointers!

Dynamic Hashing

- At some point using hashing, would like to start over.
  - Employ different hashing function, allowing a larger hashtable to begin with (the hash function’s possible outputs control the effective size before overflowing) and, ideally, better distribution of data
  - Expensive to rebuild a table in this “en-masse” context (previous actions were all one-shot actions to insert or query)

Dynamic Hashing

- Dynamic Hashing: Allow table to grow or shrink as insertions/deletions occur, not in a post-processing step

- Extendable hashing (a dynamic hashing technique):
  - Have hash function generate 32-bit integer values
  - Allocate data to buckets in table based on bit values

Extendable Hashing

- Initially, all data goes into one bucket until that bucket fills
  - Search cost is bound by size of bucket
- When bucket fills up, split based on some *x* bits of hash values (*x* = depth)
  - All records sharing the same x bit values go in same bucket
  - Typically depth increased by 1 (as needed)
- If empty bucket occurs from deleting data, could end up collapsing (shrinking) directory

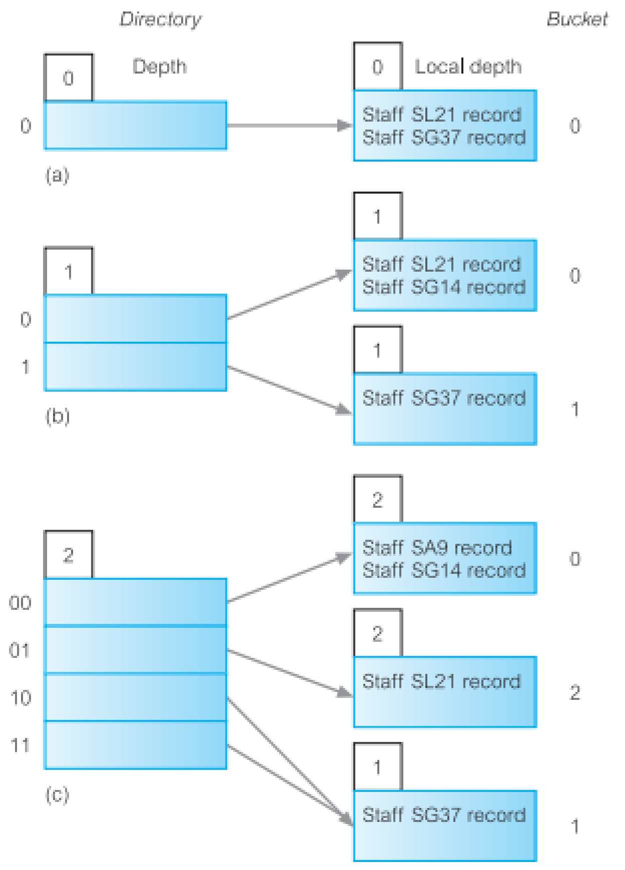


Figure F.7 Example of extendible hashing: (a) after insert of SL21 and SG37; (b) after insert of SG14; (c) after insert of SA9.

Hashing

- Hashing is not an appropriate technique for some queries….

Any suggestions of what those queries might be?

Hashing

- Hashing is not an appropriate technique for some queries….

Any suggestions of what those queries might be?

Queries based on ranges (> 1000 and < 10000)

Queries based on similarity (LIKE S%)

Queries based on anything other than the field used for hashing

Indexing

- One of the most commonly used techniques for improving efficiency of database operations is the notion of an index.

- An index is just what you think it is (as in a book index)
  - An ordered list of record identifiers, each associated with references to where the actual record exists.

Indexing

- Three primary types of indices:
  - Primary index (one of these per file/table)
    - Actual records stored in sorted order based on a key (unique) for the table; index based on the key as well
  - Clustering index (one of these per file/table, if primary not used – *why?*)
    - Actual records stored in sorted order based on a non- key for the table; index based on the non-key (meaning 1+ records with same value in index, pointing to different places)
  - Secondary index (multiple of these per file/table)
    - An index that is based an attribute that the data is not sorted on
- Note the indices themselves are always ordered (the data referenced doesn’t have to be for secondary)

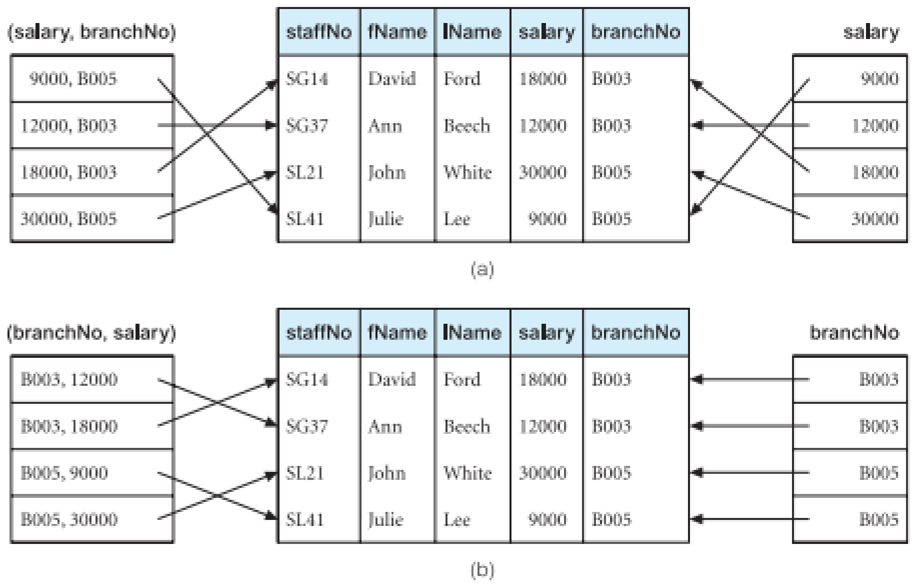
Indexing

- Index can be on composite attributes
  - Have an internal ordering (assume left-to-right based subfield ordering)
- A few example indices for the DreamHome Staff table

Which of these appear to be:
Primary indices?

Clustering indices?

Secondary indices?



(a)

(salary, branchNo)
9000, B005
12000, B003
18000, B003
30000, B005

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

salary
9000
12000
18000
30000

(b)

(branchNo, salary)
B003, 12000
B003, 18000
B005, 9000
B005, 30000

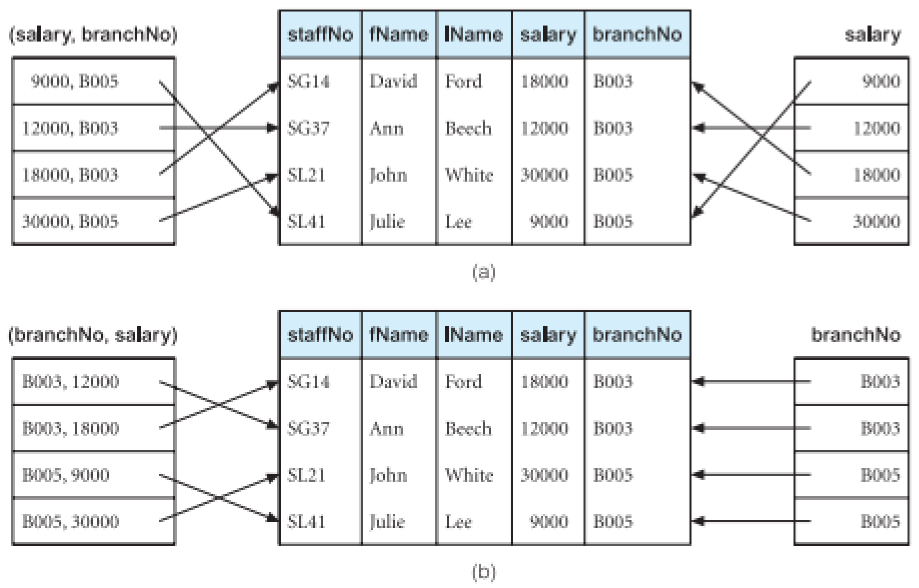
staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

branchNo
B003
B003
B005
B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Indexing

- A few example indices for the DreamHome Staff table



staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Which of these are:

Primary indices?
NONE

Clustering indices?
(branchNo, salary)
branchNo

Secondary indices?
salary
(salary, branchNo)

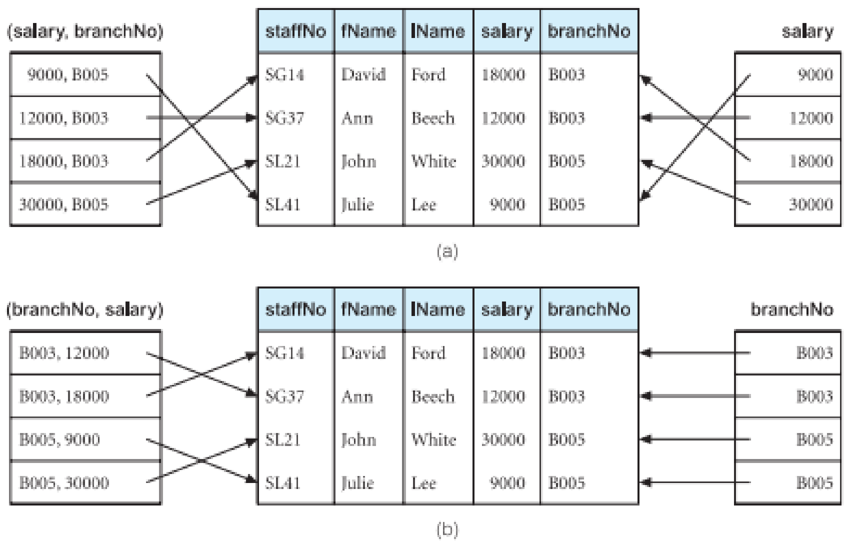
Appears to be sorted by branch ascending , and then by salary within branch descending

Indexing

- Advantages of using indices:
  - Loading into memory an index is cheaper than loading a whole original table
    - Often have as many rows in index as you have in table, especially if primary index
    - But, index usually two columns: index value, pointer/reference
  - Because of ordering of index, can stop after found value of interest

Indexing

- Advantages of using indices:
  - Indices can be used for range based queries – *why?* (i.e. salary > 10,000 & salary < 20,000)
  - *The index itself is in sorted order, so ranges are easy to work with if index exists for that attribute*



(salary, branchNo)
9000, B005
12000, B003
18000, B003
30000, B005

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

salary
9000
12000
18000
30000

(branchNo, salary)
B003, 12000
B003, 18000
B005, 9000
B005, 30000

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

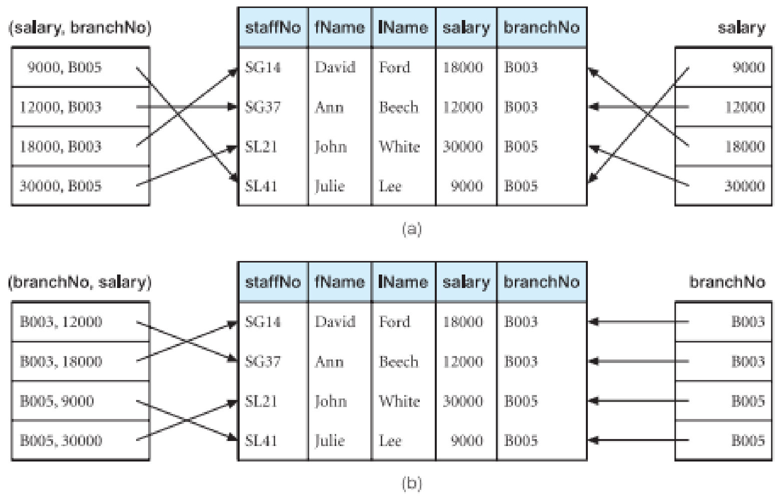
branchNo
B003
B003
B005
B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Indexing

- Advantages of using indices:
  - There are some questions (which we can formulate as queries) that can be answered using the index alone
    - *An example?*

Does anyone work at Branch BXXX?



(a)

(salary, branchNo)
9000, B005
12000, B003
18000, B003
30000, B005

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

salary
9000
12000
18000
30000

(b)

(branchNo, salary)
B003, 12000
B003, 18000
B005, 9000
B005, 30000

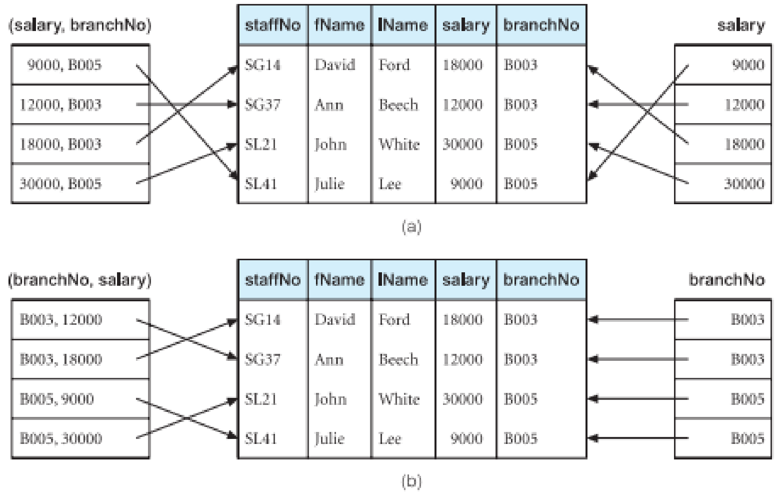
staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

branchNo
B003
B003
B005
B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Indexing

- Advantages of using indices:
  - A composite attribute index is also an index on some sub-pieces of the index
  - For an index on attributes ABC, it also acts as an index on attributes A and AB



(salary, branchNo)
9000, B005
12000, B003
18000, B003
30000, B005

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

salary
9000
12000
18000
30000

(branchNo, salary)
B003, 12000
B003, 18000
B005, 9000
B005, 30000

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

branchNo
B003
B003
B005
B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Indexing

- ☹ There’s no free lunch …
- Indices speed up access to data, but we have to pay for it somewhere else?

  – *Where do you think that is?*

Indexing

- ☹ There’s no free lunch …
- Indices speed up access to data, but we have to pay for it somewhere else?
  - We have to maintain the index when items are inserted or deleted from the table
  - Requires maintaining index value and associated pointer in sorted order
    - Insertion of a new value in sorted list: linear in size of list

Indexing

- Our goal for indexing was to effectively get rid of having to do sequential scan of data file (tables)

- Aren’t we doing now a linear search, just on a different file?
  - *Yes, though remember we can stop early!*

Indexing

- What if we introduced an index to an index?