CS 5614: (Big) Data Management Systems

B. Aditya Prakash
Lecture #4: Constraints, Storing and Indexes
Constraints in Relational Algebra and SQL
Maintaining Integrity of Data

- Data is *dirty*.
- How does an application ensure that a database modification does not corrupt the tables?

In a way—

"Hi, this is your son's school. We're having some computer trouble."

"Oh, dear—did he break something?"

"DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?"

"Oh, yes. Little Bobby tables, we call him."

"Well, we've lost this year's student records. I hope you're happy."

"And I hope you've learned to sanitize your database inputs."
Maintaining Integrity of Data

- Data is **dirty**.
- How does an application ensure that a database modification does not corrupt the tables?

- Two approaches:
  - Application programs check that database modifications are consistent.
  - Use the features provided by SQL.
Integrity Checking in SQL

- PRIMARY KEY and UNIQUE constraints.
- FOREIGN KEY constraints.
- Constraints on attributes and tuples.
- Triggers (schema-level constraints).

- How do we express these constraints?
- How do we check these constraints?
- What do we do when a constraint is violated?
Keys in SQL

- A set of attributes $S$ is a key for a relation $R$ if every pair of tuples in $R$ disagree on at least one attribute in $S$.

- Select one key to be the PRIMARY KEY; declare other keys using UNIQUE.
Primary Keys in SQL

- Modify the schema of Students to declare PID to be the key.
  - CREATE TABLE Students(
    PID VARCHAR(8) PRIMARY KEY,
    Name CHAR(20), Address VARCHAR(255));

- What about Courses, which has two attributes in its key?
  - CREATE TABLE Courses(Number integer, DeptName:
    VARCHAR(8), CourseName VARCHAR(255), Classroom
    VARCHAR(30), Enrollment integer,
    PRIMARY KEY (Number, DeptName));
**Effect of Declaring PRIMARY KEYS**

- Two tuples in a relation cannot agree on all the attributes in the key. DBMS will reject any action that inserts or updates a tuple in violation of this rule.

- A tuple cannot have a NULL value in a key attribute.
Other Keys in SQL

- If a relation has other keys, declare them using the UNIQUE keyword.
- Use UNIQUE in exactly the same places as PRIMARY KEY.

There are two differences between PRIMARY KEY and UNIQUE:
- A table may have only one PRIMARY KEY but more than one set of attributes declared UNIQUE.
- A tuple may have NULL values in UNIQUE attributes.
Enforcing Key Constraints

- Upon which actions should an RDBMS enforce a key constraint?
  - Only tuple update and insertion.
  - RDBMS searches the tuples in the table to find if any tuple exists that agrees with the new tuple on all attributes in the primary key.
  - To speed this process, an RDBMS automatically creates an efficient search index on the primary key.
  - User can instruct the RDBMS to create an index on one or more attributes
Foreign Key Constraints

- **Referential integrity constraint**: in the relation Teach (that “connects” Courses and Professors), if Teach relates a course to a professor, then a tuple corresponding to the professor must exist in Professors.

- How do we express such constraints in Relational Algebra?

- Consider the Teach(ProfessorPID, Number, DeptName) relation.

  Every non-NULL value of ProfessorPID in Teach must be a valid ProfessorPID in Professors.

- **RA**: \( \pi_{\text{ProfessorPID}}(\text{Teach}) \subseteq \pi_{\text{PID}}(\text{Professors}) \).
Foreign Key Constraints in SQL

- every non-NULL value of ProfessorPID in Teach must be a valid ProfessorPID in Professors.
- In Teach, declare ProfessorPID to be a foreign key.
- CREATE TABLE Teach(ProfessorPID VARCHAR(8) REFERENCES Professor(PID), Name VARCHAR(30) ...);
- CREATE TABLE Teach(ProfessorPID VARCHAR(8), Name VARCHAR(30) ..., FOREIGN KEY ProfessorPID REFERENCES Professor(PID));
- If the foreign key has multiple attributes, use the second type of declaration.
Requirements for FOREIGN KEYS

- If a relation R declares that some of its attributes refer to foreign keys in another relation S, then these attributes must be declared UNIQUE or PRIMARY KEY in S.

- Values of the foreign key in R must appear in the referenced attributes of some tuple in S.
Enforcing Referential Integrity

- **Three** policies for maintaining referential integrity.

- **Default policy**: reject violating modifications.

- **Cascade policy**: mimic changes to the referenced attributes at the foreign key.

- **Set-NULL policy**: set appropriate attributes to NULL.
Specifying Referential Integrity Policies in SQL

- SQL allows the database designer to specify the policy for deletes and updates independently.
- Optionally follow the declaration of the foreign key with `ON DELETE` and/or `ON UPDATE` followed by the policy: `SET NULL` or `CASCADE`.
- Constraints can be circular, e.g., if there is a one-one mapping between two relations.
- In this case, SQL allows us to defer the checking of constraints.
Constraining Attributes and Tuples

- SQL also allows us to specify constraints on attributes in a relation and on tuples in a relation.
  - Disallow courses with a maximum enrollment greater than 100.
  - A chairperson of a department must teach at most one course every semester.

- How do we express such constraints in SQL?
- How can we change our minds about constraints?
- A simple constraint: NOT NULL
  - Declare an attribute to be NOT NULL after its type in a CREATE TABLE statement.
  - Effect is to disallow tuples in which this attribute is NULL.
Attribute-Based CHECK Constraints

- Disallow courses with a maximum enrollment greater than 100.
  - This constraint only affects the value of a single attribute in each tuple.

- CREATE TABLE Courses(...
  Enrollment INT CHECK (Enrollment <= 100) ...);

- The condition can be any condition that can appear in a WHERE clause.

- CHECK statement may use a subquery to mention other attributes of the same or other relations.

- An attribute-based CHECK constraint is checked only when the value of that attribute changes.
Tuple-Based CHECK Constraints

- Tuple-based CHECK constraints are checked whenever a tuple is inserted into or updated in a relation.
- Designer may add these constraints after the list of attributes in a CREATE TABLE statement.
- A chairperson of a department teaches at most one course in any semester.

CREATE TABLE Teach(...

    CHECK ProfessorPID NOT IN
    ((SELECT ProfessorPID FROM Teach)
    INTERSECT
    (SELECT ChairmanPID FROM Departments)
    );
Modifying Constraints

- SQL allows constraints to be named.
- Use CONSTRAINT followed by the name of the constraint in front of PRIMARY KEY, UNIQUE, or CHECK.
- Can use constraint names in ALTER TABLE statements to delete constraints: say DROP CONSTRAINT followed by the name of the constraint.
- Can add constraints in an ALTER TABLE statement using ADD CONSTRAINT followed by an optional name followed by the (required) CHECK statement.
Assertions

- These are database-schema elements, like relations
- Defined by:
  - CREATE ASSERTION <name>
    CHECK (<condition>);
- Condition may refer to any relation or attribute in the database schema.
Assertions: Example

- Can’t have more courses than students (‘Pigeonhole Principle’)

CREATE ASSERTION FewStudents CHECK (
(SELECT COUNT(*) FROM Students) <=
(SELECT COUNT(*) FROM Courses)
);
Triggers: Motivation

- Assertions are powerful, but the DBMS often can’t tell when they need to be checked.
- Attribute- and tuple-based checks are checked at known times but not powerful.
- Triggers let the user decide when to check for any condition.
OK, what could have been done?

HI, THIS IS YOUR SON’S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING? IN A WAY-

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU’RE HAPPY.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Announcements

- Check updated webpage...
- More details on
  - Grading scheme
  - Homework dates
  - Project deliverables
  - Lecture Schedule
STORING DATA
DBMS Layers:

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

TODAY ➔
Leverage OS for disk/file management?

- Layers of abstraction are good ... but:
Leverage OS for disk/file management?

- Layers of abstraction are good … but:
  - Unfortunately, OS often gets in the way of DBMS
Leverage OS for disk/file management?

- DBMS wants/needs to do things “its own way”
  - Specialized prefetching
  - Control over buffer replacement policy
    - LRU not always best (sometimes worst!!)
  - Control over thread/process scheduling
    - “Convoy problem”
      - Arises when OS scheduling conflicts with DBMS locking
  - Control over flushing data to disk
    - WAL protocol requires flushing log entries to disk
Disks and Files

- DBMS stores information on disks.
  - but: disks are (relatively) VERY slow!
- Major implications for DBMS design!
Disks and Files

- Major implications for DBMS design:
  - **READ**: disk -> main memory (RAM).
  - **WRITE**: reverse
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
Why Not Store It All in Main Memory?
Why Not Store It All in Main Memory?

- **Costs too much.**
  - disk: \(~\$1/\text{Gb}\); memory: \(~\$100/\text{Gb}\)
  - High-end Databases today in the 10-100 TB range.
  - Approx 60% of the cost of a production system is in the disks.

- **Main memory is volatile.**

- **Note:** some specialized systems do store entire database in main memory.
The Storage Hierarchy

Smaller, Faster → Bigger, Slower
–Main memory (RAM) for currently used data.

–Disk for the main database (secondary storage).

–Tapes for archiving older versions of the data (tertiary storage).
Jim Gray’s Storage Latency Analogy: How Far Away is the Data?

- **Andromeda**: $10^9$ Tape, 2,000 Years
- **Disk**: $10^6$, 2 Years
- **Memory**: 100, 1.5 hr
- **On Board Cache**: 10, 10 min
- **On Chip Cache**: 2, 10 min
- **Registers**: 1, 1 min

- **Boston**
- **This Building**
- **This Room**
- **My Head**
Disks

- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - relative placement of pages on disk is important!
Anatomy of a Disk

- **Sector**
- **Track**
- **Cylinder**
- **Platter**
- **Block size** = multiple of sector size (which is fixed)
Accessing a Disk Page

- Time to access (read/write) a disk block:
  - .
  - .
  - .
Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time*: moving arms to position disk head on track
  - *rotational delay*: waiting for block to rotate under head
  - *transfer time*: actually moving data to/from disk surface
Accessing a Disk Page

- **Relative times?**
  - *seek time:*
  - *rotational delay:*
  - *transfer time:*
Accessing a Disk Page

- Relative times?
  - *seek time*: about 1 to 20msec
  - *rotational delay*: 0 to 10msec
  - *transfer time*: $< 1$ msec per 4KB page
Seek time & rotational delay dominate

- Key to lower I/O cost: reduce seek/rotation delays!
- Also note: For shared disks, much time spent waiting in queue for access to arm/controller
Arranging Pages on Disk

- “Next” block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

- Accessing ‘next’ block is cheap

- A useful optimization: pre-fetching
  - See textbook page 323
Rules of thumb...

1. Memory access much faster than disk I/O (~ 1000x)
   - “Sequential” I/O faster than “random” I/O (~ 10x)
Disk Arrays: RAID

Benefits:

- Higher throughput (via data "striping")
- Longer MTTF (via redundancy)
Recall: DBMS Layers

Queries

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

DB

TODAY →
Buffer Management in a DBMS

Page Requests from Higher Levels

(copy of a) disk page

free frame

MAIN MEMORY

DISK

DB

choice of frame dictated by replacement policy

buffer pool

Just FYI
Files

- **FILE**: A collection of pages, each containing a collection of records.

- Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)
Alternative File Organizations

Several alternatives (w/ trade-offs):

- **Heap files**: Suitable when typical access is a file scan retrieving all records.
- **Sorted Files**:
- **Index File Organizations**:
Variable length records

- SLOTTED PAGE organization - popular.
Conclusions---Storing

- Memory hierarchy
- Disks: (>1000x slower) - thus
  - pack info in blocks
  - try to fetch nearby blocks (sequentially)
- Record organization: Slotted page
TREE INDEXES
Declaring Indexes

- No standard!
- Typical syntax:

  CREATE INDEX StudentsInd ON Students(ID);

  CREATE INDEX CoursesInd ON Courses(Number, DeptName);
Types of Indexes

- **Primary:** index on a key
  - Used to enforce constraints

- **Secondary:** index on non-key attribute

- **Clustering:** order of the rows in the data pages correspond to the order of the rows in the index
  - Only one clustered index can exist in a given table
  - Useful for range predicates

- **Non-clustering:** physical order not the same as index order
Using Indexes (1): Equality Searches

- Given a value $v$, the index takes us to only those tuples that have $v$ in the attribute(s) of the index.

- E.g. (use CourseInd index)

```sql
SELECT Enrollment FROM Courses
WHERE Number = "4604" and DeptName = "CS"
```
Using Indexes (1): Equality Searches

- Given a value $v$, the index takes us to only those tuples that have $v$ in the attribute(s) of the index.

- Can use Hashes, but see next
Using Indexes (2): Range Searches

- ``Find all students with gpa > 3.0’’
- may be slow, even on sorted file
- Hashes not a good idea!
- What to do?
Range Searches

- "Find all students with gpa > 3.0’’
- may be slow, even on sorted file
- Solution: Create an ‘index’ file.

![Diagram showing data file and index file with pointers]

Prakash 2014

VT CS 5614
Range Searches

- More details:
- if index file is small, do binary search there
- Otherwise??

Prakash 2014
B-trees

- the most successful family of index schemes (B-trees, B+-trees, B*-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced “n-way” search trees
B-trees

- Eg., B-tree of order $d=1$:
B - tree properties:

- each node, in a B-tree of order d:
  - Key order
  - at most n=2d keys
  - at least d keys (except root, which may have just 1 key)
  - all leaves at the same level
  - if number of pointers is k, then node has exactly k-1 keys
  - (leaves are empty)
Properties

- “block aware” nodes: each node is a disk page
- $O(\log (N))$ for everything! (ins/del/search)
- typically, if $d = 50 - 100$, then 2 - 3 levels
- utilization $\geq 50\%$, guaranteed; on average 69%
Queries

- Algo for exact match query? (eg., ssn=8?)
JAVA animation

- http://slady.net/java/bt/
Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

- Algo for exact match query? (eg., ssn=8?)
Algo for exact match query? (eg., ssn=8?)
Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

- what about range queries? (eg., $5<\text{salary}<8$)
- Proximity/ nearest neighbor searches? (eg., salary $\sim 8$)
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)
Queries

- what about range queries? (e.g., $5 < \text{salary} < 8$)
- Proximity/nearest neighbor searches? (e.g., salary $\sim 8$)
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary \sim 8 )
Variations

- How could we do even better than the B-trees above?
B+ trees - Motivation

- B-tree – print keys in sorted order:
B+ trees - Motivation

- B-tree needs back-tracking – how to avoid it?
B+ trees - Motivation

- Stronger reason: for clustering index, data records are scattered:

```
<6

1 3

>6

6 9

<9

7

>9

13
```
Solution: B+ - trees

- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level
- (vital, for clustering index!)
B+ trees
B+ trees

Index Pages

Data Pages
B+ trees

- More details: next (and textbook)
- In short: on split
  - at leaf level: COPY middle key upstairs
  - at non-leaf level: push middle key upstairs (as in plain B-tree)
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
Inserting a Data Entry into a B+ Tree

- Find correct leaf L.
- Put data entry onto L.
  - If L has enough space, done!
  - Else, must split L (into L and a new node L2)
    • Redistribute entries evenly, copy up middle key.

- parent node may overflow
  - but then: push up middle key. Splits “grow” tree; root split increases height.
Example B+ Tree – Inserting 30*
Example B+ Tree – Inserting 30*

Root

2* 3* 5* 7*
14* 16*
19* 20* 22* 23*
24* 27* 29* 30*
Example B+ Tree - Inserting 8*
Example B+ Tree - Inserting 8*

No Space
Example B+ Tree - Inserting 8*

So Split!
Example B+ Tree - Inserting 8*

So Split!

And then push middle UP
Example B+ Tree - Inserting 8*

Final State
Example B+ Tree - Inserting 21*

Root

5 13 17 24

2* 3* 5* 7* 8* 14* 16* 19* 20* 22* 23* 24* 27* 29*

24* 27* 29*
Example B+ Tree - Inserting 21*

Root is Full, so split recursively
Example B+ Tree: Recursive split

• Notice that root was also split, increasing height.
Example: Data vs. Index Page Split

- leaf: ‘copy’
- non-leaf: ‘push’
- why not ‘copy’ @ non-leaves?
Same Inserting 21*: The Deferred Split

Note this has free space. So...
Inserting 21*: The Deferred Split

LEND keys to sibling, through PARENT!
Inserting 21*: The Deferred Split

Shorter, more packed, faster tree

Prakash 2014
VT CS 5614
Insertion examples for you to try

Insert the following data entries (in order): 28*, 6*, 25*
After inserting 28*, 6*

After inserting 25*
Answer...

After inserting 25*
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L underflows
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
      - update parent
      - and possibly merge, recursively
Deletion from B+Tree
Example: Delete 19* & 20*

Deleting 19* is easy:

- Dele>ng 19*
- Dele>ng 20*
- re-distribu>on
  - (notice: 27 copied up)

Root

- 17
- 24 30
- 33 34 38 39
- 24* 27* 29*
- 20* 22*

- 14* 16*
- 5* 7* 8*
- 2* 3*
... And Then Deleting 24*

Praka: • Must merge leaves: OPPOSITE of insert
... And Then Deleting 24*

... but are we done??

Praca: • Must merge leaves: OPPOSITE of insert
... Merge Non-Leaf Nodes, Shrink Tree

![Tree Diagram]

Prakash 2014

VT CS 5614
Example of Non-leaf Re-distribution

- Tree is shown below during deletion of 24*.
- Now, we can re-distribute keys.
After Re-distribution

- need only re-distribute ‘20’; did ‘17’, too
- why would we want to re-distribute more keys?
Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?
Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?
- ‘lazy deletions’ - in fact, some vendors just mark entries as deleted (~ underflow), – and reorganize/compact later
Recap: main ideas

- on overflow, split (and ‘push’, or ‘copy’)
  – or consider deferred split

- on underflow, borrow keys; or merge
  – or let it underflow...
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = $2 \times 100 \times 0.67 = 134$

- Typical capacities:
  - Height 4: 1334 = 312,900,721 entries
  - Height 3: 1333 = 2,406,104 entries
B+ Trees in Practice

- Can often keep top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB
B+ trees with duplicates

- Everything so far: assumed unique key values
- How to extend B+-trees for duplicates?
  - Alt. 2: <key, rid>
  - Alt. 3: <key, {rid list}>
- 2 approaches, roughly equivalent
B+ trees with duplicates

- approach#1: repeat the key values, and extend B+ tree algo’s appropriately - eg. many ‘14’ s
B+ trees with duplicates

- approach#1: subtle problem with deletion:
- treat rid as part of the key, thus making it unique

Prakash 2014
B+ trees with duplicates

- approach#2: store each key value: once
- but store the \{rid list\} as variable-length field (and use overflow pages, if needed)
B+trees in Practice

- prefix compression;
- bulk-loading;
- ‘order’
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic’; can often compress them.
Bulk Loading of a B+ Tree

- In an empty tree, insert many keys
- Why not one-at-a-time?
  - Too slow!
Bulk Loading of a B+ Tree

- Initialization: Sort all data entries
- scan list; whenever enough for a page, pack
- <repeat for upper level>
Bulk Loading of a B+ Tree

![Diagram of a B+ Tree with data entry pages not yet in the tree.](image)
A Note on `Order`

- Order (d) concept replaced by physical space criterion in practice (`at least half-full`).
- Why do we need it?
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3)).
A Note on `Order’

- Many real systems are even sloppier than this: they allow underflow, and only reclaim space when a page is completely empty.
- (what are the benefits of such ‘slopiness’?)
Conclusions

- B+tree is the prevailing indexing method
- Excellent, $O(\log N)$ worst-case performance for ins/del/search; (~3-4 disk accesses in practice)
- guaranteed 50% space utilization; avg 69%
Conclusions

- Can be used for any type of index: primary/secondary, sparse (clustering), or dense (non-clustering)
- Several fine-extensions on the basic algorithm
  - deferred split; prefix compression; (underflows)
  - bulk-loading
  - duplicate handling