Final Reference Sheet — CS 143 — Fall 2015

Ky-Cuong L. Huynh

December 6, 2015

0.1 Relational Algebra

WHERE C is equivalent to the select operator σ_C, with C incorporating comparisons ($>$, $\leq$, $\neq$) and connectives ($\wedge, \vee, \neg$). Note the use of set semantics, which has NO DUPLICATES.

SELECT A FROM R is equivalent to the project operator $\Pi_A(R)$. List attributes to keep from R.

FROM R, S is equivalent to the cross (Cartesian) product $R \times S$, which generates every possible tuple concatenation. Example: $(a_1, a_2) \times (b_1, b_2, b_3) = ((a_1, b_1), (a_1, b_2), \dots, (a_2, b_1), \dots)$. Use this to compare tuples within a single relation, with the columns renamed.

$TableNameAST1$ is equivalent to $\rho_{T1}(TableName)$, the rename operator. Use it alias tables to allow prefixing of attributes.

WHERE $R1.A = R2.A$ is equivalent to the natural join operator $R_1 \bowtie R_2$, which is shorthand for $\sigma_{R_1.A=R_2.A}(R1 \bowtie R_2)$. It enforces equality on common attributes, but leaves the unshared ones alone, with all tuples concatenated horizontally. We assume only one copy of common attributes is kept.

WHERE $R1.A = R2.B$ is one example of a theta join, which is a generalization of natural join: $R_1 \bowtie_C R_2 = \sigma_C(R_1 \times R_2)$.

The union operator $R \cup S$ requires renames/projections to match schema for combination (with no duplicates in result). Example: $\Pi_{name}(Student) \cup \rho_{R_{name}}(\Pi_{instructor}(Class))$

The set difference operaotr $R - S$ requires exact schema matches. Often, we lose information to match schemas, and then join again to restore that information. Example: titles of courses with at least one student taking them: $\Pi_{title}((\Pi_{dept,cnum,title}(Class) - \Pi_{dept,cnum,sec}(Enroll)) \bowtie Class)$.

The intersect operator can be defined as $R \cap S = R - (R - S)$.

The division operator is defined for relations $R(A, B)$ and $S(B)$. R/S is the set of all $a \in R.A$ such that $\langle a, b \rangle \in R$ for every $b \in S$. Formally, $R/S = \{ a : a \in R.A \wedge \langle a, b \rangle \in R \forall b \in S \}$. Informally, if R stores student-class pairs, and S is all CS classes, then R/S is the students who take all CS classes. Alternatively: $R/S = \Pi_A(R) - \Pi_A((\Pi_A(R) \times S) - R)$.

0.2 SQL

By default, bag semantics, except for the set operators. To eliminate duplicates, use DISTINCT. To keep duplicates for the sets, use $UNIONALL$, etc.

Try writing a difficult query's complement.

Tricks for checking if SQL/relational algebra statements are equivalent: set vs. bag semnatics (set operators, including EXCEPT, are always set semantics), $A = A$ excludes NULL tuples, $<$ vs. $\leq$. In general, WRITE OUT EXAMPLES.

INTERSECT, UNION, EXCEPT all follow set semantics.

General SQL statement: `SELECT attributes, aggregates FROM relations WHERE conditions GROUP BY attributes HAVING conditions on aggregates ORDER BY attributes, aggregates`

Evaluation order: FROM → WHERE → GROUP BY → HAVING → ORDER BY → SELECT.

For strings: % is any length string, with _ being one character.

Set membership: IN and NOT IN. We can compare using ALL (for all), and SOME (there exists). We use EXISTS to check for a non-empty sub-query result: EXISTS(SELECT FROM WHERE).

Example: `CREATE TABLE Course(dept CHAR(2) NOT NULL, cnum INTEGER, instructor VARCHAR(30), PRIMARY KEY(dept, cnum, sec), UNIQUE(dept, cnum, instructor));`

0.3 Aggregates

SUM, AVG, COUNT, MIN, MAX. Use DISTINCT to avoid duplicates: SELECT COUNT(DISTINCT sid) FROM Enroll WHERE dept='CS' for "number of students taking CS classes". But if a filtering attribute is not unique, then use set membership: `SELECT AVG(GPA) FROM Students WHERE sid IN (SELECT sid FROM Enroll WHERE dept='CS')` for "the average GPA of students taking CS classes."

For GROUP BY, the SELECT must only be for attributes that have a single value in each group or aggregates. For example: SELECT sid FROM Student GROUP BY age makes no sense; how does an age-group have an SID. Conditions on aggregates are enforced by HAVING, but queries can be rewritten to not have them in general. Example: `SELECT sid FROM Enroll WHERE dept='CS' GROUP BY sid HAVING COUNT(*) >= 2` gives all the students taking two or more CS classes.

We can display tuples in a sorted order. By default, it's listed in ascending order (ASC). Example: `SELECT sid, GPA FROM Student ORDER BY GPA DESC, sid ASC`. No change to results returned.

COUNT <attr> ignores NULL values; only COUNT(*) includes them.

0.4 NULL Complexities

Arithmetic operations with NULL inputs return NULL. Arithmetic comparison with NULL inputs return Unknown. For set operations, NULL is treated like any other value. To check for nullity: IS NULL or IS NOT NULL. Aggregates are computed ignoring NULL values, except for COUNT(*). Empty (including null values) inputs to aggregates give output NULL, except for COUNT, which gives 0.

Only True tuples are returned from queries. We introduce Unknown because something like `GPA * 100/4 > 90` for a NULL GPA will have a result that depends on GPA's eventual value. If we made them all False, then negations/complements would return NULL tuples alongside actually false ones.

Unknown AND True =¿ Unknown. Unknown OR True =¿ True. Unknown OR false =¿ Unknown. NOT Unknown =¿ Unknown.

0.5 Bag Semantics

Multisets allow duplicate elements, but order continues to remain irrelevant. Not all set rules hold; check them manually.

Union keeps all: $\{a, a, b\} \cup \{a, b, c\} = \{a, a, a, b, b, c\}$.

Intersection takes the minimum of the instances on the LHS vs the RHS: $\{a, a, a, b, c\} \cap \{a, a, b\} = \{a, a, b\}$.

Difference subtracts the number of instances in RHS from LHS; if negative, then zero instances in the result: $\{a, a, b, b\} - \{a, b, b, c\}$.

0.6 Insertion, Updates, Deletion

We can insert literal tuples: `INSERT INTO Enroll VALUES (301, 'CS;, 201, 01), (201, 'EE', 203, 03)` Or we can insert the results of a query: `INSERT INTO Honors SELECT * FROM Student WHERE GPA > 3.7`.

Update specify new values for specific attributes: `UPDATE R SET A1 = V1, A2 = V2 WHERE condition`

We delete based on those matching a predicate: `DELETE FROM Student WHERE sid NOT IN (...)` To delete a whole table's entries: `DELETE From Table`

0.7 Integrity, Triggers, & Constraints

Example: `sid INTEGER REFERENCES Student(sid)` For multiple attrs: `FOREIGN KEY(dept, cnum, sec) REFERENCES Class(dept, cnum, sec)`.

If E.A references S.A, then violating inserts/updates to E (referencing table) are always rejected, but violations to S (referenced table) can be handled gracefully. Always insert/update S first. If the S-tuple is updated/deleted, we disallow the statement by default. Otherwise, we can specify ON DELETE and/or ON UPDATE SET NULL or SET DEFAULT. E.A value will be set to null or default. CASCADE will propagate the change in S.A to E.A: the referencing tuple will be updated/deleted.

CHECK checks the values of tuples upon insert/update: `GPA real CHECK(0 <= GPA AND GPA <= 4.0)`, and can be complex with subqueries.

Triggers watch for events that meet conditions and execute some action in response: `CREATE TRIGGER CS143Req AFTER INSERT ON Student REFERENCING NEW ROW as new_row FOR EACH ROW WHEN (new_row.gpa >= 3.0) INSERT INTO Enroll VALUES (new_row.sid, 'CS', 143, 1);`

0.8 Views & Authorization

Tricks for checking if a statement is allowed: substitute the query name with its definition, and see if the user is allowed to carry out that query. Note that multiple lists in a FROM clause imply a need for SELECT privileges.

Creating a view: `CREATE View ViewName(A1, A2) AS Query`. The attribute list is optional and aliases the attributes. Views can be used to store complex queries, and their tuples are dynamically generated by rewriting the view reference as their defining query. Views can be made atop other views. Users must have at least SELECT privileges on base tables. If the user loses privileges for the base tables, the views may be automatically dropped. There is no standard for table-level privileges.

Modifications to views propagate down to the underlying tables. Missing columns are filled with the default value or NULL. If neither is possible, the modification is rejected. For views to be updateable, they must be selecting on a single table T, not use DISTINCT, and their subqueries in WHERE must not refer to T. Those attributes in T not projected to the view can be NULL or default. Absolutely no aggregation.

Insertions/updates into a view WITH CHECK OPTION are rejected if the new tuple is not still in the view. If we want to drop a view and other views reference it, then we must specify DROP CASCADE (drop anything that references the doomed view), or DROP RESTRICT (the drop fails if the doomed view has dependents).

We can grant privileges for SELECT, Insert(A1, A2), Update(A1, A2), DELETE. `GRANT (priv) ON R TO (users) [WITH GRANT OPTION]`. Use ALL PRIVILEGES if needed. Users can be PUBLIC. All privileges begin with the DBA. Full privileges on particular tables are given to those who created them.

When we revoke privileges, we can do `REVOKE (priv) ON R FROM (users) [CASCADE | RESTRICT]`. Default is RESTRICT (reject revoke if privileges passed on). CASCADE takes out all privileges passed on. Many systems take care of multiple paths to a user too.

0.9 Disks and Files

Access time = (seek time)+ (rotational delay) + (transfer time). Average 15 ms. Random I/O incurs this often, and SSDs are very fast at this. Seek time = time to move disk head between tracks. Average: 10 ms, full is 20 ms; track-to-track is 1 ms. Rotational delay = time for right sector to spin around under read head. Full: 10 ms; avg: 5 ms. Transfering one second takes 0.001 ms. Calculate as time for one track divided by number of sectors for time/sector. Burst transfer rate is (RPM/60) * (sectors/track) * (bytes/sector).

RAID 0: stripe parts of files across drives. RAID 1: stripe files across drives, but have an independent copy of each stripe. RAID 5: stripe, with parity bits distributed across drives (RAID 4 concentrates them into one drive; professor miswrote 4 for 5). Allows single drive to be reconstructed if it's lost.

Tuples can be stored with empty space (unspanned) or we can pack parts of them across blocks (spanned). The former is simpler but wastes up to 50% space. Variable-length tuples can have ther maximum space reserved (waste indeterminate, possibly high amount of space), or we can pack them tightly, and track their location using a slotted page. The block keeps a header that points to the internal positions of each tuple. An index just points to the start of the header. Very long tuples are often stored separately from short ones, giving us fast access for short attrs. To maintain a sequential file, we can rearrange tuples upon insert, or keep a linked list. If we run out of space, we allocate an overflow page.

0.10 Indexing

A primary (clustering) index is one whose index entries' ordering matches the order of the indexed tables' entries, i.e., $10 \to 10$, $20 \to 20$. A dense index has a (key,pointer) index entry pair for every record. We assume that the tuples are in a sequential file. We build our indices based on its search keys. Even though this gives us the same number of entries, each pair is smaller, and the whole index fits into RAM.

A sparse index has (key,pointer) for every block, and points to the first entry. We find the largest index entry less than our index search key, jump to that block, then scan through.

A secondary (non-clustering) index does not have its entries in the same order as the table, but must be dense on its first level. For example, we index on a non-search-key attribute. We can have multi-level indexes. All of this is called ISAM (indexed sequential access method).

Insertion into ISAM: follow search and go to index entry: empty (key,pair) block? Fill it. Else, try moving the neighbor down to the next block. If full, allocate an overflow page. Performance degrades over time.

B+ trees are n-ary, balanced, and only their leaves point to actual tuples. Inner nodes are signways to the leaves. They are characterized by n, the number of pointers per node. Leaf nodes have all except the last pointing to tuples (the last one point to the the next leaf). Non-leaf nodes have pointers to the nodes in the level below. [<]23[≤<]56[≥]

All nodes have up to n pointers and $n-1$ keys. The root node has at least 2 pointers and 1 key. Non-leaf nodes have at least $\lceil(n+1)/2\rceil$ pointers and $\lceil(n-1)/2\rceil$ keys. Leaf nodes have at least $\lceil n/2\rceil$ pointers and $\lceil n/2\rceil - 1$ keys.

To search B+ trees, we follow the points according to the value(s) stored in the non-leaf nodes, and follow the pointer to the tuples once we're at leaf (based on boolean or precomputed depth).

To insert into B+ trees, we have four cases. No-overflow: go to right leaf node and add (key, pointer). If a leaf overflows, then allocate another leaf, split the entries over (move new over, or one of old), then copy the first (leftmost) key of the new node to the (rightmost pointer of the parent of the old leaf. Non-leaf overflow also splits and propagates upwards. If we need a new root, then split and move up the root key, creating a new root, with left and right becoming new nodes.

0.11 Join Algorithms

Index join is the fastest if an index already exists and its look-up cost and number of matching tuples is not too high, as we only have to load all of one table, and selectively load from the other. Sort-Merge is the second fastest, if the data is already sorted, as we just merge. Otherwise, hash join tends to be the best, as its buckets allow us to nested-loop behavior when joining. Nested-loop join (with block nesting) is usable for small tables, and its smaller table should always be in the outer loop. Costs are given for $b_R < b_S$, where b is the number of blocks, $|S|$ is the number of tuples, and M is the size of memory, in blocks.

Nested loop (with both optimizations): $b_R + (b_S \cdot \lceil \frac{b_R}{M-2} \rceil)$

Sort-merge (last term only if already sorted; number of sorted runs is $\lceil b_R/M \rceil$, as we can read in all into memory for sorting): $2b_R(\lceil \log_{M-1}(b_R/M)\rceil + 1) + 2b_S(\lceil \log_{M-1}(b_S/M)\rceil) + (b_R + b_S)$

Hash join: $2(b_R + b_S)\left\lceil \log_{M-1}\left(\frac{b_R}{M-2}\right)\right\rceil + (b_R + b_S)$

Index join (where C is the average index look-up cost, there are J matching tuples in S for every R tuple, of which there are $|R|$ many: $b_R + |R| \cdot (C + J)$

0.12 Functional Dependencies and Decomposition

$u[X]$ is the values of the attributes X for tuple u. There is a functional dependency $X \to Y$ if for any $u_1, u_2 \in R$, if $u_1[X] = u_2[X]$, then $u_1[Y] = u_2[Y]$. This is trivially true when $Y \subseteq X$, non-trivial otherwise, and completely non-trivial if no overlap at all.

Closure (set of all attrs determined by X) computation: (1) Start with $X^+ = X$. (2) Until there is no change in X^+, see if there is a $Y \to Z$ and $Y \subseteq X^+$, and if so, then add Z to X^+.

X is a key of R iff: (1) $X \to$ all attrs of R ($X^+ = R$) and (2) no subset of$X$ fulfills that, i.e., X is minimal. A decomposition of a table $R(X,Y,Z) \Rightarrow R_1(X,Y), R_2(X,Z)$ is lossless if $X \to Y$ or $X \to Z$.

0.13 Boyce-Codd Normal Form (BCNF)

R is in BCNF iff for *every* $X \to Y$, X contains a key. Algorithm: for *any* table R, if non-trivial $X \to Y$ and X lacks a key: (1) Compute X^+ and (2) Decompose R into $R_1(X^+)$ and $R_2(X,Z)$, where Z is every attribute but X^+, i.e., $Z = R - X^+$.

0.14 Multi-valued Dependencies and Fourth Normal Form (4NF)

There is a multi-valued dependency $X \twoheadrightarrow Y$ if for every tuple $u, v \in R$ that share values on the attributes X, i.e., $u[X] = v[X]$, then: $w[X] = u[X] = v[X]$; $w[Y] = u[Y]$; $w[Z] = v[Z]$, where Z is all attributes in R except X and Y. That is, if two tuples agree on X, we can swap their Y values, and the two resulting tuples should also exist.

The complementation rule says that swapping Z is no different: if $x \twoheadrightarrow Y$, then also $X \twoheadrightarrow Z$. $X \twoheadrightarrow Y$ is trivial if (1) $Y \subseteq X$ or (2) $X \cup Y = R$.

A relation R is in 4NF if for every non-trivial FD $X \to Y$ and every MVD $X \twoheadrightarrow Y$, X contains a key. It removes redundancies from MVDs.

Decomposition for 4NF: (0) Normalize into BCNF first. Algorithm: for *any* table R, if non-trivial $X \twoheadrightarrow Y$ and X lacks a key: (1) Decompose R into $R_1(X,Y)$ and $R_2(X,Z)$, where Z is all attributes but (X,Y), i.e., $Z = R - (X \cup Y)$.

0.15 Transactions

ACID: Atomicity (all-or-nothing operations), Consistency (data is always in a consistent state, before and after transactions), Isolation (concurrent transactions have an effect equivalent to being executed in sequential isolation), Durability (committed changes to data are never lost).

COMMIT commits a set of statements; a BEGIN may precede them. ROLLBACK undoes all changes back to the previous commit; multiple rollbacks are not possible on most systems.

AUTOCOMMIT mode OFF has transactions beginning whenever data is read/written and ends them when a COMMIT or ROLLBACK is encountered. Having this mode ON automatically ends a transaction after every single SQL statement.

To ensure these properties, we maintain a log, that can be replayed. Most systems use write-ahead logging, where planned changes are written down before they're executed. Changes to the log are managed using locks. Specifically, the (rigorous) 2-phase locking protocol allows us to achieve execution orders for our SQL statements that cause no issues. These are called conflict-free serializable schedules.

0.16 E/R Model

Don't forget to include a key attribute for each entity set. To translate to tables: (1) Have a table for each entity set, with all associated attrs. (2) Table for each relationship set, with keys borrowed from each connected set as our attributes. Cardinality dtermines that: one-to-one has either one has key; many-to-many has both together as a key; many-to-one has many side as the key. (3) Weak entity sets have attributes and borrowed key from parent entity set.

(4) For inheritance, we can create: (i) a super table with all attributes of parent and subclasses: Student(name, addr, country, fellowship). (ii) a table for each subclass, with key from parent: Student, ForeignStudent, HonorsStudent. (iii) a table for parent and every logical combination of subclasses, e.g., ForeignHonorsStudent. The first is simple, but takes up a lot of storage; the second is compact but joins a lot for queries; the third avoids redundancy and is comprehensive, but is fragmented., but takes up a lot of storage; the second is compact but joins a lot for queries; the third avoids redundancy and is comprehensive, but is fragmented.