Exercise Sheet 2, 2026

6.0 VU Advanced Database Systems

Julian Luckenberger (12222154)

May 12, 2026

Exercise 1 (Distributed Joins) *[3 Punkte]*

General Assumptions

- **Join Cardinality:** The expected number of records for $Authors \bowtie Books$ is calculated as $4000 \times 50000 \times \frac{1}{9500} \approx 21.053$ records.
- **Outer Join Cardinality:** The expected number of records for $Libraries \bowtie (Authors \bowtie Books)$ is $10000 \times 21053 \times \frac{1}{4000} \approx 52.633$ records.
- **Join Record Sizes:** Unless projecting, the size of a joined record is the sum of the two base record sizes (e.g., $450 + 400 = 850$ bytes).
- **Semi-Join Estimates:** As instructed, the worst-case estimate $\min(|A \bowtie B|, |A|)$ is used for result sizes.

(a) Computations for Standard Strategies (Query 1)

The goal is to compute the full join and send the output $\pi_{\text{name, title}}(Authors \bowtie_{\text{authid=author}} Books)$ to Site 4. In these standard strategies, whole tuples are transferred during intermediate steps.

Base Table Sizes

- **Authors (Site 1):** 4.000 records $\times$ 450 bytes = 1.800.000 bytes.
- **Books (Site 2):** 50.000 records $\times$ 400 bytes = 20.000.000 bytes.
- **Final Join Result Size:** 21.053 records $\times$ 850 bytes = 17.895.050 bytes.

Strategy Costs

1. **Join at Site 4:** Send Authors (1.800.000 bytes) and Books (20.000.000 bytes) to Site 4.
   Total: 21.800.000 bytes.

2. **Join at Site 2:** Send Authors to Site 2 (1.800.000 bytes). Compute the join and send the result to Site 4 (17.895.050 bytes).
   Total: 19.695.050 bytes.

3. **Join at Site 1:** Send Books to Site 1 (20.000.000 bytes). Compute the join and send the result to Site 4 (17.895.050 bytes).
   Total: 37.895.050 bytes.

4. **Semi-join Authors → Books:** Send `authid` from Authors ($4.000 \times 12 = 48.000$ bytes) to Site 2. The semi-join yields $\min(21053, 50000) = 21.053$ Books records. Send matching Books ($21.053 \times 400 = 8.421.200$ bytes) back to Site 1. Send the final join result to Site 4 (17.895.050 bytes).
   Total: 26.364.250 bytes.

5. **Semi-join Books → Authors:** Send `author` from Books ($50.000 \times 10 = 500.000$ bytes) to Site 1. The semi-join yields $\min(21053, 4000) = 4.000$ Authors records. Send matching Authors ($4.000 \times 450 = 1.800.000$ bytes) to Site 2. Send the final join result to Site 4 (17.895.050 bytes).
   Total: 20.195.050 bytes.

Optimal Strategy for (a): Strategy 2 minimizes the communication cost at 19.695.050 bytes.

(a*) Computations with Early Projection (Query 1)

By projecting out unneeded columns as early as possible, the size of the transferred records changes significantly.

New Projected Sizes

- **Projected Authors:** Needs `authid` (12 bytes) and `name` (58 bytes) = 70 bytes per record. Total size = 280.000 bytes.
- **Projected Books:** Needs `author` (10 bytes) and `title` (30 bytes) = 40 bytes per record. Total size = 2.000.000 bytes.
- **Projected Join Result:** Only needs `name` (58 bytes) and `title` (30 bytes) = 88 bytes per record. Total size = $21.053 \times 88 = 1.852.664$ bytes.

Adapted Strategy Costs

1*. Send projected Authors (280,000) and Books (2,000,000) to Site 4.
Total: 2.280.000 bytes.

2*. Send projected Authors to Site 2 (280.000). Join and send projected result to Site 4 (1.852.664).
Total: 2.132.664 bytes.

3*. Send projected Books to Site 1 (2.000.000). Join and send projected result to Site 4 (1.852.664).
Total: 3.852.664 bytes.

4*. Send `authid` to Site 2 (48.000). Send matching projected Books back to Site 1 ($21.053 \times 40 = 842.120$). Send result to Site 4 (1.852.664).
Total: 2.742.784 bytes.

5*. Send `author` to Site 1 (500.000). Send matching projected Authors to Site 2 ($4.000 \times 70 = 280.000$). Send result to Site 4 (1.852.664).
Total: 2.632.664 bytes.

Optimal Strategy for (a*): Strategy 2* is now the optimal execution plan at 2.132.664 bytes.

(b) Generalizing to >2 Relations (Query 2)

For Query 2, we execute $Libraries \bowtie_{\text{libid=library}} \pi_{\text{name,library}}(Authors \bowtie_{\text{authid=author}} Books)$ with the result needed at Site 4.

Inner Join Calculation (T_{inner})

First, we determine the attributes needed for the inner join $T_{inner} = \pi_{\text{name,library}}(Authors \bowtie Books)$:

- T_{inner} **size:** Requires `name` (58 bytes) and `library` (10 bytes) = 68 bytes per record.
- **Total** T_{inner} **size:** 21.053 records × 68 bytes = 1.431.604 bytes.

Using the optimal path from (a*), we can materialize T_{inner} at Site 2 for 280.000 bytes (by sending projected Authors to Site 2).

Justification for Skipping Calculations

The final join combines T_{inner} (68 bytes) and Libraries (750 bytes). The total record size for the final relation is $68 + 750 = 818$ bytes. With an estimated final cardinality of 52.633 records, the final result size is approximately **43.053.794 bytes** (∼43 MB).

Because this result is massive, any strategy that computes the final join at Site 2 or Site 3 and transfers the finished relation to Site 4 will automatically incur a transfer cost of over 43.000.000 bytes. This safely rules out computing the final outer join anywhere except Site 4.

Furthermore, applying a semi-join between T_{inner} and Libraries yields no reduction because the worst-case semi-join estimates dictate that all records survive ($\min(52633, 10000) = 10000$ and $\min(52633, 21053) = 21053$). Semi-joins here would only add communication overhead without reducing the payload.

The Optimal Generalized Strategy

1. Project Authors to `authid` and `name` at Site 1 and send to Site 2 (280.000 bytes).
2. Join at Site 2 to create T_{inner} and send T_{inner} directly to Site 4 (1.431.604 bytes).
3. Send the full Libraries table (10.000 records $\times$ 750 bytes = 7.500.000 bytes) from Site 3 to Site 4.
4. Compute the final join entirely at Site 4.

Total Optimal Cost: $280.000 + 1.431.604 + 7.500.000 =$ **9.211.604 bytes.**

Exercise 2 (Cost of MapReduce) *[2 Punkte]*

(a) Trace of the MapReduce Algorithm ($k = 2$)

First, we establish the initial set of edges E extracted from the bipartite user-item interaction graph:

- u_1 connects to i_1, i_2, i_3, i_4
- u_2 connects to i_1, i_2, i_5
- u_3 connects to i_2, i_3, i_4
- u_4 connects to i_3, i_5

Stage 1: Reversing the Index

- **Mapper 1 Output:**
  $(i_1, u_1), (i_1, u_2), (i_2, u_1), (i_2, u_2), (i_2, u_3), (i_3, u_1), (i_3, u_3), (i_3, u_4), (i_4, u_1), (i_4, u_3), (i_5, u_2), (i_5, u_4)$
- **Reducer 1 Output:**
  Groups the users by their connected items:
  $(i_1, [u_1, u_2])$
  $(i_2, [u_1, u_2, u_3])$
  $(i_3, [u_1, u_3, u_4])$
  $(i_4, [u_1, u_3])$
  $(i_5, [u_2, u_4])$

Stage 2: Finding Common Items

- **Mapper 2 Output:**
  Generates user pairs $(u_1 < u_2)$ for every shared item in the value lists:
  $((u_1, u_2), 1)$ *(from i_1)*
  $((u_1, u_2), 1), ((u_1, u_3), 1), ((u_2, u_3), 1)$ *(from i_2)*
  $((u_1, u_3), 1), ((u_1, u_4), 1), ((u_3, u_4), 1)$ *(from i_3)*
  $((u_1, u_3), 1)$ *(from i_4)*
  $((u_2, u_4), 1)$ *(from i_5)*

- **Reducer 2 Output:**
  Sums the similarity values for each user pair:
  $((u_1, u_2), 2)$
  $((u_1, u_3), 3)$
  $((u_1, u_4), 1)$
  $((u_2, u_3), 1)$
  $((u_2, u_4), 1)$
  $((u_3, u_4), 1)$

Stage 3: Top-K Filtering ($k = 2$)

- **Mapper 3 Output:**
  Emits the symmetric similarity scores for both users in a pair:
  $(u_1, (u_2, 2)), (u_2, (u_1, 2))$
  $(u_1, (u_3, 3)), (u_3, (u_1, 3))$
  $(u_1, (u_4, 1)), (u_4, (u_1, 1))$
  $(u_2, (u_3, 1)), (u_3, (u_2, 1))$
  $(u_2, (u_4, 1)), (u_4, (u_2, 1))$
  $(u_3, (u_4, 1)), (u_4, (u_3, 1))$

- **Reducer 3 Output:**
  Groups candidates by user and retains only the top $k = 2$ highest scores. In cases of tied scores, we resolve by picking the user with the lower index:
  For u_1: Top candidates are u_3 (score 3) and u_2 (score 2).
  For u_2: Top candidate is u_1 (score 2). Ties exist for the second spot (u_3 and u_4 both have 1). We arbitrarily pick u_3.
  For u_3: Top candidate is u_1 (score 3). Ties exist for the second spot (u_2 and u_4 both have 1). We arbitrarily pick u_2.
  For u_4: All candidates (u_1, u_2, u_3) are tied with a score of 1. We arbitrarily pick u_1 and u_2.

  Final Emitted Output:
  $((u_1, u_3), 3), ((u_1, u_2), 2)$
  $((u_2, u_1), 2), ((u_2, u_3), 1)$
  $((u_3, u_1), 3), ((u_3, u_2), 1)$
  $((u_4, u_1), 1), ((u_4, u_2), 1)$

(b) Cost Measures Analysis

Communication cost: Total number of input elements sent to reducers
Replication rate: Average number of key-value pairs created from each mapper input
Reducer size: Maximum size of the value list for a single key.

Stage 1

- **Communication cost:** $|E|$. The mapper emits exactly one key-value pair for every edge in the bipartite graph.

- **Replication rate:** 1. Each input edge translates to exactly one output pair.
- **Reducer size:** $\max_{v \in V} \deg(v)$. The keys for Reducer 1 are items (v), and the value list contains all users connected to that item. The largest list corresponds to the item with the highest degree.

Stage 2

- **Communication cost:** $\sum_{v \in V} \binom{\deg(v)}{2}$. For every item v, Mapper 2 emits a key-value pair for every unique combination of 2 users in its value list.
- **Replication rate:** $\frac{1}{|V|} \sum_{v \in V} \binom{\deg(v)}{2}$. The input to Mapper 2 consists of $|V|$ elements (the item-user lists). The replication rate is the total number of output pairs divided by the number of input elements.
- **Reducer size:** $\max_{u_1, u_2 \in U} |CI(u_1, u_2)|$. The keys for Reducer 2 are user pairs (u_1, u_2). The value list contains a 1 for every item they share, so the length of the list is exactly the number of common items between them.

Stage 3

- **Communication cost:** $2 \cdot |P|$, where $|P|$ is the number of distinct user pairs with a similarity score greater than zero (i.e., $|P| = |\{(u_1, u_2) \mid CI(u_1, u_2) \neq \emptyset\}|$).
- **Replication rate:** 2. Mapper 3 takes a single user pair as input and emits two symmetric key-value pairs (one keyed by u_1 and one keyed by u_2).
- **Reducer size:** $\max_{u \in U} |S(u)|$. The key for Reducer 3 is a single user u. Its value list contains all other users v that share at least one item with u, which by definition is the set $S(u)$.

Exercise 3 (MapReduce Algorithms) *[2 Punkte]*

(a) Relational Algebra Expression

Given the expression $F_1(X, Y, Z) = \gamma_b SUM(a)(\sigma_{a>b}(X) - \pi_{a,b}(Y \ltimes Z))$, we can decompose this into a sequence of three MapReduce jobs to handle the semi-join/projection, the selection/set difference, and the final aggregation.

Job 1: Semi-Join and Projection $\pi_{a,b}(Y \ltimes_{b,c} Z)$

This job finds all tuples in Y that have a matching tuple in Z on the common attributes (b, c), and projects the result down to (a, b).

- **Map:**
  - **Input:** A split of relations $Y(a, b, c)$ and $Z(b, c, d)$.
  - For each tuple (a, b, c) in Y, Emit key (b, c) and value ("Y", a).
  - For each tuple (b, c, d) in Z, Emit key (b, c) and value ("Z").
- **Reduce:**

  - **Input:** Key (b, c), and a list of values from Y and Z.
  - Check if the list of values contains at least one "Z" tag.
  - If a "Z" is present, iterate through the list. For every ("Y", a) found, Emit key (a, b) and value NULL.

Job 2: Selection $\sigma_{a>b}(X)$ and Set Difference

This job applies the selection condition on X and subtracts the tuples generated by Job 1.

- **Map:**
  - **Input:** A split of relation $X(a, b)$ and the output tuples from Job 1.
  - For each tuple (a, b) in X: evaluate the selection condition. If $a > b$, Emit key (a, b) and value ("X").
  - For each tuple (a, b) from Job 1: Emit key (a, b) and value ("Y_Z").
- **Reduce:**
  - **Input:** Key (a, b), and a list of tags.
  - Check the contents of the value list. If the list contains "X" but **does not** contain "Y_Z", the tuple survives the set difference.
  - For surviving tuples, Emit key b and value a. (This prepares the data for the final group-by stage).

Job 3: Grouping and Aggregation $\gamma_b SUM(a)$

This final job groups the surviving pairs by b and calculates the sum of a.

- **Map:**
  - **Input:** Key b and value a from Job 2.
  - Identity Mapper: Emit key b and value a.
- **Reduce:**
  - **Input:** Key b, and a list of values $[a_1, a_2, \ldots, a_n]$.
  - Compute the sum $S = \sum_{k=1}^{n} a_k$.
  - Emit key b and value S.

(b) Matrix Min-Max Normalization

Assuming that a single row of matrix M fits into the memory of a single reducer, we can compute the min-max normalized matrix M' in a single MapReduce round.

- **Map:**
  - **Input:** A split of matrix M, represented as tuples (i, j, M_{ij}).
  - For each tuple, Emit key i (the row index) and value (j, M_{ij}).

- **Reduce:**
  - **Input:** Key i, and a list of values representing the elements of that row: $[(j_1, M_{i1}), (j_2, M_{i2}), \dots]$.
  - **Pass 1:** Iterate over the list of values to find the minimum and maximum values for the row. Let these be $\min(M_i)$ and $\max(M_i)$.
  - **Pass 2:** Iterate over the list of values a second time. For each (j, M_{ij}), compute the normalized value M'_{ij}:
    - If $\max(M_i) = \min(M_i)$, let $M'_{ij} = 0$.
    - Otherwise, let $M'_{ij} = \frac{M_{ij} - \min(M_i)}{\max(M_i) - \min(M_i)}$.
  - Emit key (i, j) and value M'_{ij} (or emit as a full tuple (i, j, M'_{ij})).

Exercise 4 (Spark) *[5 Punkte]*

(b) Spark Processing Model Analysis

(i) Stages and Boundaries (Ignoring Optimization Blocks)

In Spark's processing model, a job is divided into stages by **wide dependencies** (operations that require a network shuffle, such as `join` without a known partitioner, `groupByKey`, or `reduceByKey`). Narrow dependencies (like `map` and `filter`) are pipelined together within the same stage.

Assuming no optimization blocks are used:

- **`user_scores` Job:**
  - *Stage 1:* Reading `posts`, mapping, and filtering into `high_score_posts`.
  - *Stage 2:* Reading `votes` and mapping.
  - *Stage 3:* The `join()` of `high_score_posts` and `votes`. Since neither has a known partitioner, this is a wide transformation requiring a shuffle. This stage maps the joined output.
  - *Stage 4:* The `reduceByKey()` operation creates another wide dependency/shuffle boundary to compute the final `user_scores`.
- **`tag_scores` Job:** Follows the exact same 4-stage lineage as above, triggering a full re-evaluation of the data from the source files.
- **`user_comment_scores` Job:** Similar to the above, but Stage 2 reads/maps the `comments` RDD instead of `votes`, before joining and reducing.

(ii) Impact of Optimization Blocks on Application 1

Application 1 executes an action (`take(5)`) on all three final RDDs.

- **Optimization Block 1 (`partitionBy(4).cache()`): Improves performance.** By using `partitionBy(4)` on both `high_score_posts` and `votes`, Spark assigns them a known `HashPartitioner`. This means that when `join()` is called, the data is already co-located on the executors, making the join into a *narrow dependency* instead of a wide dependency and eliminating an expensive network shuffle. Furthermore, `cache()` on `high_score_posts` ensures the massive posts dataset is only filtered/mapped once, rather than re-evaluated for both `post_votes` and `post_comments`.

- **Optimization Block 2 (`post_votes.cache()`): Improves performance.** `post_votes` is a shared parent RDD for both `user_scores` and `tag_scores`. Caching it prevents Spark from recalculating the entire thing again bottom up when `tag_scores.take(5)` triggers its action.

(iii) Impact of Optimization Blocks on Application 2

Application 2 only executes actions on `user_scores` and `tag_scores` (ignoring `user_comment_scores`).

- **Optimization Block 1:** Does **not improve performance**. As `high_score_posts` is technically only used in the `post_votes` branch for Application 2, the `partitionBy(4)` still successfully eliminates the expensive network shuffle during the join with `votes`, however the `partitionBy(4)` itself needs a shuffle here so this basically canceles out.
- **Optimization Block 2:** Still **improves performance**. Because both `user_scores` and `tag_scores` are still evaluated, caching the shared `post_votes` RDD remains crucial to avoid computing the join twice.