Temporal Aggregation and Join
SL05

- Temporal aggregation
  - Span, instant, moving window aggregation
  - Aggregation tree, balanced tree, bucket algorithm
- Temporal joins
  - Temporal Cartesian product, temporal joins
  - Evaluation algorithms

Acknowledgements: I am indebted to Michael Böhlen for providing me the slides.
Query Processing

- One of the most important tasks of a DBMS is to figure out an **evaluation strategy** for high level (query) statements.
- Query processing is a 3-step process

1. **Parsing and translation**
   - Check syntax and verify relations.
   - Translate the query into an equivalent relational algebra expression.

2. **Optimization**
   - Generate an optimal (lowest cost) evaluation plan for the query.

3. **Evaluation**
   - The execution engine takes an (optimal) evaluation plan, executes that plan, and returns the answers to the query.

- Cost is measured as **number of (block) IOs**.
Temporal Aggregation
  ▶ Forms of temporal aggregation
  ▶ Aggregation tree algorithm
  ▶ Balanced tree algorithm
  ▶ Bucket algorithm

Temporal Joins
  ▶ Block nested loop
  ▶ Sort merge
  ▶ Partition join
  ▶ Index join
Snapshot Aggregation

- **Snapshot aggregation** transforms an argument relation into a summary result relation
  - Result relation can be a single tuple (a single aggregation group) or several tuples (multiple aggregation groups)

- Two-step process:
  1. Partitioning the argument relation into groups of tuples with identical values for one or more attributes (→ aggregation groups)
  2. Apply one or more aggregate functions (e.g., count, sum) to each of these groups in turn.

- **Example**: Non-temporal aggregation queries in SQL
  - Compute the average salary of employees grouped by department.
    ```sql
    SELECT Dept, AVG(Salary)
    FROM Employee
    GROUP BY Dept
    ```
Temporal Aggregation

- Additionally use time dimension to group argument tuples.
- **Temporal grouping**: Process where the timeline is partitioned over time and argument tuples are grouped over these time partitions.
  - The aggregate values are computed over each of these groups in turn.
- Two types of temporal grouping:
  - **Span grouping**: Timeline is partitioned into fixed-length intervals, e.g., month or year.
    - Independent of argument tuples
  - **Instant grouping**: Timeline is partitioned into instants/chronons.
    - Results are reported over so-called constant intervals, i.e., sequences of instants with the same argument tuples (and consequently the same aggregate values).
    - Constant intervals depend on the argument tuples
    - Result relation might be larger than argument relation (up to twice as large)
Different forms of temporal aggregation have been proposed:

- **Instant temporal aggregation**
  - computes a nontemporal aggregate at each point in time; considers tuples valid at the point in time; data lineage can be considered

- **Moving-window temporal aggregation**
  - computes a nontemporal aggregation at each point in time; considers time points at nearby time points as well

- **Span temporal aggregation**
  - computes a nontemporal aggregation of data that falls into predefined time ranges, e.g., a year or month
Different Forms of Temporal Aggregation/1

- **Instant temporal aggregation (ITA)**
  - computes aggregate results for each time point $t$ over all tuples that are valid at $t$
    - an aggregation group is associated with each point in time
    - the aggregation group contains all tuples that overlap the point in time
    - an aggregation function is applied to each aggregation group and produces a single value
  - coalesces consecutive tuples with identical aggregate values into constant intervals.
    - Lineage information might be considered or not
Different Forms of Temporal Aggregation/2

- **Qita**: For each month and department, what is the number of contracts?

\[
\begin{align*}
\mathbf{r}_1 &= (\text{Jan},140,\text{DB},1200,[1,12]) \\
\text{EMP} \mathbf{r}_2 &= (\text{Ann},141,\text{DB},700,[1,5]) \\
\mathbf{r}_3 &= (\text{Ann},150,\text{DB},700,[6,15]) \\
\mathbf{r}_4 &= (\text{Tom},143,\text{AI},2000,[4,9]) \\
\end{align*}
\]

\[
\begin{align*}
Q_{\text{ita}} &= \boxed{(\text{DB},2,[1,5])} \\
&\quad \boxed{(\text{DB},2,[6,12])} \\
&\quad \boxed{(\text{DB},1,[13,15])} \\
&\quad \boxed{(\text{AI},1,[4,9])} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>CID</th>
<th>Dept</th>
<th>Sal</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>140</td>
<td>DB</td>
<td>1200</td>
<td>[1,12]</td>
</tr>
<tr>
<td>Ann</td>
<td>141</td>
<td>DB</td>
<td>700</td>
<td>[1,5 ]</td>
</tr>
<tr>
<td>Ann</td>
<td>150</td>
<td>DB</td>
<td>700</td>
<td>[6,15]</td>
</tr>
<tr>
<td>Tom</td>
<td>143</td>
<td>AI</td>
<td>2000</td>
<td>[4,9 ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>Cnt</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>2</td>
<td>[1,5]</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>[6,12]</td>
</tr>
<tr>
<td>DB</td>
<td>1</td>
<td>[13,15]</td>
</tr>
<tr>
<td>AI</td>
<td>1</td>
<td>[4,9]</td>
</tr>
</tbody>
</table>

TSDB15, SL05
Moving-window temporal aggregation (MWTA)

- also termed cumulative temporal aggregation
- a time window is used to determine aggregation groups
- computes aggregate results for each time point $t$ over all tuples that are valid in $[t - w, t]$
- other possibilities for time windows exist, e.g., $[t - w, t + w']$
- coalesces consecutive tuples as in ITA
- differs from ITA in the definition of the aggregation groups
Different Forms of Temporal Aggregation

- $Q_{\text{mwta}}$: For each month and department, how many contracts have been in effect during this month and the preceding two months?

\[
\begin{align*}
  r_1 &= (\text{Jan}, 140, \text{DB}, 1200, [1,12]) \\
  \text{EMP} \ r_2 &= (\text{Ann}, 141, \text{DB}, 700, [1,5]) \\
  r_3 &= (\text{Ann}, 150, \text{DB}, 700, [6,15]) \\
  r_4 &= (\text{Tom}, 143, \text{AI}, 2000, [4,9])
\end{align*}
\]

$Q_{\text{mwta}} = (\text{DB,}2, [1,5]) \quad (\text{DB,}3, [6,7]) \quad (\text{DB,}2, [8,14]) \quad (\text{DB,}1, [15,17]) \quad (\text{AI,}1, [4,11])$

<table>
<thead>
<tr>
<th>Name</th>
<th>CID</th>
<th>Dept</th>
<th>Sal</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>Jan</td>
<td>140</td>
<td>DB</td>
<td>1200</td>
</tr>
<tr>
<td>r₂</td>
<td>Ann</td>
<td>141</td>
<td>DB</td>
<td>700</td>
</tr>
<tr>
<td>r₃</td>
<td>Ann</td>
<td>150</td>
<td>DB</td>
<td>700</td>
</tr>
<tr>
<td>r₄</td>
<td>Tom</td>
<td>143</td>
<td>AI</td>
<td>2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>Cnt</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>2</td>
<td>[1,5]</td>
</tr>
<tr>
<td>DB</td>
<td>3</td>
<td>[6,7]</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>[8,14]</td>
</tr>
<tr>
<td>DB</td>
<td>1</td>
<td>[15,17]</td>
</tr>
<tr>
<td>AI</td>
<td>1</td>
<td>[4,11]</td>
</tr>
</tbody>
</table>

TSDB15, SL05
Span temporal aggregation (STA)

- the time domain is partitioned into predefined intervals
- computes the aggregate results for each user-specified time period (span).
- an aggregation group includes all tuples that overlap the span
- often fixed granularities are allowed as spans: years, months, days
- timestamps of result tuples (spans) are specified without considering the argument tuples
Q_{sta}: For each half-year period and department, what is the number of contracts?

\[ r_1 = (\text{Jan}, 140, \text{DB}, 1200, [1,12]) \]
\[ r_2 = (\text{Ann}, 141, \text{DB}, 700, [1,5]) \]
\[ r_3 = (\text{Ann}, 150, \text{DB}, 700, [6,15]) \]
\[ r_4 = (\text{Tom}, 143, \text{AI}, 2000, [4,9]) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>CID</th>
<th>Dept</th>
<th>Sal</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>140</td>
<td>DB</td>
<td>1200</td>
<td>[1,12]</td>
</tr>
<tr>
<td>Ann</td>
<td>141</td>
<td>DB</td>
<td>700</td>
<td>[1,5]</td>
</tr>
<tr>
<td>Ann</td>
<td>150</td>
<td>DB</td>
<td>700</td>
<td>[6,15]</td>
</tr>
<tr>
<td>Tom</td>
<td>143</td>
<td>AI</td>
<td>2000</td>
<td>[4,9]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>Cnt</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>3</td>
<td>[1,6]</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>[7,12]</td>
</tr>
<tr>
<td>DB</td>
<td>1</td>
<td>[13,18]</td>
</tr>
<tr>
<td>AI</td>
<td>1</td>
<td>[1,6]</td>
</tr>
<tr>
<td>AI</td>
<td>1</td>
<td>[7,12]</td>
</tr>
<tr>
<td>AI</td>
<td>0</td>
<td>[13,18]</td>
</tr>
</tbody>
</table>

TSDB15, SL05

A. Dignös
Kline and Snodgrass (1995) proposed the so-called aggregation tree algorithm for the evaluation of temporal instant aggregation:

- **Step 1:** Incrementally construct the aggregation tree which manages the constant intervals and partial aggregation results.
- **Step 2:** Perform a depth-first search of the tree and compute the final aggregation values at the leaves.

- Each node in the aggregation tree stores
  - a start and end time of an interval;
  - an aggregate state value, i.e., partial result of aggregate function over this interval.

- Each leaf node encodes a constant interval.
The aggregation tree can be constructed in an **incremental** way (for one or several aggregate function).

The initial aggregation tree contains a single node valid from 0 to $\infty$ (entire timeline) and a neutral value for the aggregate state value.

For each new tuple:
- Search for the constant interval(s) containing the start and end time of the tuple
- If the time point falls between the boundaries of the constant interval
  - Split the interval into two new constant intervals
  - Update the aggregate state value
- If the time point coincides with a boundary of the constant interval
  - Update the aggregate state value
- For all internal nodes that are completely covered by the argument tuple
  - Update the aggregate state value and stop to search further in this sub-tree
Example: Employee relation of a company

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richard</td>
<td>40K</td>
<td>18</td>
<td>∞</td>
</tr>
<tr>
<td>Karen</td>
<td>45K</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>Nathan</td>
<td>35K</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>Nathan</td>
<td>37K</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Assume a time domain from 0 to ∞.
Query: Compute the number of tuples valid over each constant interval
a. Initially, a single node valid from 0 to \( \infty \) has a count of 0

b. Add the tuple \((Richard, 40K, 18, \infty)\): The start time 18 requires to split the root node

c. Add the tuple \((Karen, 45K, 8, 20)\): The start time and the end time require a node split
d. Add the tuple \((Nathan, 35K, 7, 12)\) and \((Nathan, 37K, 18, 21)\)
Aggregation Tree Algorithm - Result Tuples

- **Producing result tuples**
  - Traverse the aggregation tree in depth-first search order
  - Keep track of the aggregate state value, i.e., accumulate additive values
  - At each leaf node, write out the aggregate value with the constant interval at the leaf

- **Example:** At leaf [8, 12] we get: $0 + 0 + 1 + 1 = 2$

Result

<table>
<thead>
<tr>
<th>Count</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
Aggregation Tree Algorithm - Evaluation

- Requires **only one scan** of the argument relation.
- Average complexity to construct the aggregation tree: $O(n \log n)$, where $n$ is the number of argument tuples
- Worst case complexity for construction: $O(n^2)$
  - If input tuples are sorted in chronological order, the tree ends up in a linear list
- Scan to produce sequenced aggregation result for all time points is $O(n)$. (Negligible compared to construction cost.)
- Lookup of an aggregation value at a time point is $O(n)$
- Scan or lookup of aggregate values in the aggregation tree is $O(n)$. (Negligible compared to construction cost.)
- Good if the aggregation tree fits into memory.
The balanced tree algorithm is built on the following two ideas:

- Timestamp sorting (basic idea)
- Creating a balanced tree (optimization of timestamp sorting; no duplicate time points; incremental construction of sorted timestamps)

**Timestamp sorting for the count aggregation**

1. Load the entire tuples in main memory
2. Extract the timestamps from the tuples and add to each time a tag that indicates whether the timestamp is a starting (S) or an ending (E) timestamp
3. Sort the timestamps (together with the tags) in increasing order
4. Scan the sorted timestamps, getting started with a counter initialized to zero
   - At each timestamp, the counter is incremented by the number of start tags at that time, and it is decremented by the number of end tags at that time.
Example: Compute the time-varying number of tuples for the following Employee relation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richard</td>
<td>46,000</td>
<td>Accounting</td>
<td>18</td>
<td>31</td>
</tr>
<tr>
<td>Karen</td>
<td>45,000</td>
<td>Shipping</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>Nathan</td>
<td>35,000</td>
<td>Marketing</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>Nathan</td>
<td>38,000</td>
<td>Accounting</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

Result

<table>
<thead>
<tr>
<th>Count</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Timestamp sorting to compute the count aggregate function.
Balanced Tree Algorithm

- **Balanced tree algorithm** to store the timestamps and the number of starting and ending tuples.

```
Algo: BalancedTree(r)

\( T \leftarrow \) empty balanced tree;

\textbf{foreach} tuple \( t \in r \) \textbf{do}

\hspace{1em} \textbf{if} \( t.\text{start\_time} = n.\text{ts} \) for any node \( n \) in \( T \) \textbf{then}

\hspace{2em} \( n.\text{no\_starts} \leftarrow +1; \)

\hspace{1em} \textbf{else}

\hspace{2em} Insert a new node \( n' \) with \( n'.\text{ts} = t.\text{start\_time} \) into \( T \);

\hspace{2em} \( n'.\text{no\_starts} \leftarrow 1; \)

\hspace{1em} \textbf{if} \( t.\text{end\_time} = n.\text{ts} \) for any node \( n \) in \( T \) \textbf{then}

\hspace{2em} \( n.\text{no\_ends} \leftarrow +1; \)

\hspace{1em} \textbf{else}

\hspace{2em} Insert a new node \( n' \) with \( n'.\text{ts} = t.\text{end\_time} \) into \( T \);

\hspace{2em} \( n'.\text{no\_ends} \leftarrow 1; \)
```
Balanced Tree Algorithm/4

- **Example:** Compute the time-varying number of tuples using the balanced tree algorithm.
  
  a. Tree after inserting the start time of the first Employee tuple *(Richard, 46, 000, Accounting, 18, 31)*
  
  b. Tree after inserting node with time 21 and before it is balanced by red-black tree operation.
  
  c. Tree after it is balanced by red-black tree operation.

▶ Inorder traversal of tree to compute the aggregation results
▶ e.g., $1 + 1 = 2$ for the time interval $[8, 12]$
Balanced Tree Algorithm - Evaluation

- **Complexity**
  - Requires only **one scan** of the relation
  - Construction complexity: \( O(n \log n) \) (worst case)
  - Lookup complexity: \( O(n) \)
  - Complexity of producing sequenced aggregation result: \( O(n) \)

- **Limitations**
  - Works also for **sum** and **avg** aggregation function.
    - For the **sum** function store at each node the sum of the attribute values of the starting/ending tuples
    - For **avg** function store the pair \((\text{sum}, \text{count})\)
  - Not applicable for **min** and **max** function
    - The balanced tree does not keep track of the life span of the tuples, which is required for **min** and **max**
    - Extensions are too costly
Bucket Algorithm

- An algorithm for **large scale temporal aggregation**
  - Argument relation does not fit into main memory
  - Only a small constant number of database scans are acceptable
- Basic idea:
  1. Do a partitioning of the timeline and assign the tuples to corresponding buckets
     - Long tuples belong to several buckets and need special treatment
  2. Compute the temporal aggregates for each bucket independently
- Similar to hash join algorithm
First approach: Assign a tuple to all buckets it overlaps.

Bucket $B_1 = \{t_1, t_4\}$
Bucket $B_2 = \{t_3, t_4\}$
Bucket $B_3 = \{t_2, t_3, t_4\}$
Bucket $B_4 = \{t_2, t_3, t_4\}$

Leads to considerable duplication of data, especially for long-lived tuples.
Bucket Algorithm - Assign Tuples to Buckets/2

- **Second approach:**
  - Suppose that the life span of a tuple \( t \) overlaps buckets \( B_i, B_{i+1}, \ldots, B_j \)
  - Assign a tuple only to the bucket where its start and end timestamps lie, i.e., to \( B_i \) and \( B_j \)
  - A **meta array** is used to aggregate the data over the intermediate buckets, i.e., \( B_{i+1}, \ldots, B_{j-1} \).
    - The size of the meta array is equal to the total number of buckets

\[
\begin{align*}
\text{Bucket } B_1 &= \{t_1, t_4\} \\
\text{Bucket } B_2 &= \{t_3\} \\
\text{Bucket } B_3 &= \{t_2\} \\
\text{Bucket } B_4 &= \{t_2', t_3', t_4'\}
\end{align*}
\]

Meta Array for count aggregation = \( \{0, 1, 2, 0\} \)
Bucket Algorithm - Computing Aggregates/3

- **Computing** the aggregates

1. Perform the temporal aggregate operation on each bucket independently (using any temporal aggregation algorithm).
2. Combine aggregation values in the meta array (\(\{0,1,2,0\}\)) with the aggregation results from corresponding buckets.
3. Merge/Coalesce adjacent buckets at the boundaries if the aggregation values are equal.

![Diagram of the bucket algorithm](image)
Bucket Algorithm - Computing Aggregates/4

Algo: BucketAlgorithm(r)

\( I_B \leftarrow \text{time interval for each bucket } ((T_{\text{max}} - T_{\text{min}})/N_B); \)

\textbf{foreach} tuple \( t \in r \) \textbf{do}

\hspace{1em} start_bucket \( \leftarrow (t.\text{start}_\text{time} - T_{\text{min}})/I_B; \)

\hspace{1em} end_bucket \( \leftarrow (t.\text{end}_\text{time} - T_{\text{min}})/I_B; \)

\hspace{1em} Insert \( t \) into a bucket \( B_{\text{start}_\text{bucket}} \);

\hspace{1em} \textbf{if} start_bucket \( \neq \) end_bucket \textbf{then}

\hspace{2em} Insert \( t' \) into a bucket \( B_{\text{end}_\text{bucket}} \);

\hspace{1em} \textbf{for} \( i = \text{start}_\text{bucket} + 1 \) \textbf{to} \( \text{end}_\text{bucket} - 1 \) \textbf{do}

\hspace{2em} Update \( \text{meta}_\text{array}[i] \);

\hspace{1em} \textbf{for} \( i = 0 \) \textbf{to} \( N_B - 1 \) \textbf{do}

\hspace{2em} Perform temporal aggregation on the bucket \( B_i \);

\hspace{2em} Combine the scalar value of \( \text{meta}_\text{array}[i] \) to the bucket \( B[i] \);

\hspace{2em} Merge the bucket boundary with \( B_{i-1} \) as needed;
Bucket Algorithm - Evaluation

- Requires **three accesses** to the database (provided that one bucket and the meta-array fit into main memory)
  - 2 reads and 1 write
- Add the complexity of temporal aggregation algorithm for each bucket
- Inherits properties of aggregation algorithms for the aggregation of each bucket.
- Overhead for managing and finding buckets must be considered.
Table of Contents

- Temporal Aggregation
  - Forms of temporal aggregation
  - Aggregation tree algorithm
  - Balanced tree algorithm
  - Bucket algorithm

- Temporal Joins
  - Block nested loop
  - Sort merge
  - Partition join
  - Index join
Evaluation of Joins

- Joins are arguably the most important relational operator
  - Efficient implementation is of paramount importance for the overall efficiency of a query processor
- Conventional techniques optimize joins with equality predicates
- In temporal databases the problem is more pronounced
  - Inequality predicates are prevalent
  - Time dimension may significantly increase the size of the database
Example

- Employee relation with employees and departments they work for
- Manages relation with managers who supervise those departments

<table>
<thead>
<tr>
<th>EmpName</th>
<th>Dept</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ron</td>
<td>Ship</td>
<td>[1,5]</td>
</tr>
<tr>
<td>George</td>
<td>Ship</td>
<td>[5,9]</td>
</tr>
<tr>
<td>Ron</td>
<td>Mail</td>
<td>[6,10]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>MgrName</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ed</td>
<td>[3,8]</td>
</tr>
<tr>
<td>Ship</td>
<td>Jim</td>
<td>[7,15]</td>
</tr>
</tbody>
</table>
Temporal Natural Join

Setup:

\[ R = (A_1, \ldots, A_n, C_1, \ldots, C_k, T_s, T_e) \]
\[ S = (B_1, \ldots, B_m, C_1, \ldots, C_k, T_s, T_e) \]

- Explicit attributes \( A_i, B_i, C_i \)
- Join attributes: \( C = \{C_1, \ldots, C_k\} \)
- \( T_s, T_e \) are the timestamp start and end attributes
- Shorthand: \( T = [T_s, T_e], A = \{A_1, \ldots, A_n\}, B = \{B_1, \ldots, B_n\} \)
- \( r, s \) are instances of \( R, S \)

Definition: Temporal natural join

\[ r \ltimes^T s = \{ z^{(n+m+k+2)} | \exists x \in r \exists y \in s ( x[C] = y[C] \land z[A] = x[A] \land z[B] = y[B] \land z[C] = y[C] \land z[T] = \text{intsct}(x[T], y[T]) \land z[T] \neq \emptyset) \} \]
Evaluation Algorithms for Temporal Joins

- Nested loop paradigm
  - Compare all pairs of tuples from the input relations

- Sort-merge paradigm
  - Sort the input relations first
  - Merge the sorted runs, where qualifying tuples are matched

- Partitioning paradigm
  - Divide the input tuples into buckets using the join attributes
  - Corresponding buckets contain all tuples that could possibly match
  - Buckets are constructed to best utilize the available main memory buffer space

- Index-based paradigm
  - Use indexes on the join attributes to locate matching tuples efficiently
  - Preexisting index or built on the fly
**Nested-Loop Join**

**Algo: NestedLoop**(\(r, s\))

\(result \leftarrow \emptyset;\)

**foreach block** \(b_r \in r\) **do**

read\((b_r)\);

**foreach block** \(b_s \in s\) **do**

read\((b_s)\);

**foreach tuple** \(x \in b_r\) **do**

**foreach tuple** \(y \in b_s\) **do**

\[\begin{align*}
\text{if } x[C] &= y[C] \text{ and } \text{intsect}(x[T], y[T]) \neq \emptyset \text{ then } \\
&\quad z[A] \leftarrow x[A]; \quad z[B] \leftarrow y[B]; \quad z[C] \leftarrow x[C]; \\
&\quad z[T] \leftarrow \text{intsect}(x[T], y[T]); \\
&\quad result \leftarrow result \cup \{z\};
\end{align*}\]

**return** \(result;\)

- \(r\) is the outer relation, \(s\) is the inner relation
- Quadratic cost
Sort-Merge-Based Join/1

- **Sort-merge-based join** evaluation works in two phases
  - **Phase 1:** The input relations \( r \) and \( s \) are sorted on the join attributes
  - **Phase 2:** Results are produced by simultaneously scanning \( r \) and \( s \), merging tuples that match

- **Complication**
  - Join attributes may not be key attributes \( \rightarrow \) multiple tuples may have identical join attribute values.
  - An \( r \) tuple may join with many \( s \) tuples
    - This sequence of \( s \) tuples is called a **scan**
  - Relation \( s \) has to be “**backed up**” to ensure that all possible matches are found.
Sort-Merge-Based Join/2

- Data structure to store **state information**

```plaintext
structure state
    integer current_block;
    integer current_tuple;
    integer first_block;
    integer first_tuple;
    block tuples;
```

- `current_block` and `current_tuple` indicate the current tuple in the scan
- `first_block` and `first_tuple` indicate the state at the beginning of a scan
- `tuples` stores the current block in main memory

- Maintained for each of the two relations
- Particularly needed to “back up” the inner relation
Initialize the state of a scan

**Algo:** `initState(relation, state)`

- `state.current_block ← 1;`
- `state.current_tuple ← 0;`
- `state.first_block ← ⊥;`
- `state.first_tuple ← ⊥;`
- `seek(relation, state.current_block);`
- `state.tuples ← read_block(relation);`
Sort-Merge-Based Join/4

- Advance the scan of the argument relation and point to the next tuple in the sorted relation
  - If the current block has been exhausted, then the next block is read

Algo: advance(relation, state)

if state.current_tuple = MAX_TUPLES then
  state.tuples ← read_block(relation);
  state.current_block ← state.current_block + 1;
  state.current_tuple ← 1;
else
  state.current_tuple ← state.current_tuple + 1;

- Return the next tuple in the scan

Algo: currentTuple(state)

return state.tuple[state.current_tuple];
Revert the current block and tuple counters to their last values

```
Algo: backUp(relation, state)
if state.current_block \neq state.first_block then
    state.current_block \leftarrow state.first_block;
    seek(relation, state.current_block);
    state.tuples \leftarrow read_block(relation);
state.current_tuple \leftarrow state.first_tuple;
```

Mark the beginning of a scan in the state

```
Algo: markScanStart(state)
state.first_block \leftarrow state.current_block;
state.first_tuple \leftarrow state.current_tuple;
```
Sort-Merge-Based Join

**Algo:** SortMergeJoin\((r, s, C)\)

\[ r' \leftarrow \text{sort}(r, C); \quad s' \leftarrow \text{sort}(s, C); \]
\[
\text{initState}(r', \text{outer\_state}); \quad \text{initState}(s', \text{inner\_state});
\]
\[ x'[C] \leftarrow \bot; \quad \text{result} \leftarrow \emptyset; \]
\[
\text{advance}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_tuple}(\text{inner\_state});
\]

**for** \(i \leftarrow 1 \text{ to } |r'| \text{ do} \)

\[ \text{advance}(r', \text{outer\_state}); \quad x \leftarrow \text{current\_tuple}(\text{outer\_state}); \]
\[ \text{if } x[C] = x'[C] \text{ then} \]
\[ \text{backUp}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_Tuple}(s', \text{inner\_state}); \]
\[ x'[C] \leftarrow x[C]; \]
\[ \text{while } x[C] > y[C] \text{ do} \]
\[ \text{advance}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_Tuple}(\text{inner\_state}); \]
\[ \text{markScanStart}(\text{inner\_state}); \]
\[ \text{while } x[C] = y[C] \text{ do} \]
\[ \text{if } \text{intsct}(x[T], y[T]) \neq \emptyset \text{ then} \]
\[ z[A] \leftarrow x[A]; \quad z[B] \leftarrow y[B]; \quad z[C] \leftarrow x[C]; \quad z[T] \leftarrow \text{intsct}(x[T], y[T]); \]
\[ \text{result} \leftarrow \text{result} \cup \{z\}; \]
\[ \text{advance}(s', \text{inner\_state}); \]

**return** \( \text{result} \);
Partition-Based Join

Partition-based join evaluation works in two phases

- **Phase 1:** The input relations are partitioned on the join attributes into **buckets**
  - Typically a hash function is used
  - Both relations are partitioned with the same hash function → parallel sets of buckets
  - Only tuples in corresponding buckets can match

- **Phase 2:** Compare tuples in corresponding buckets and produce result tuples
Algo: PartitionJoin($r, s, C$)

\[
\text{result} \leftarrow \emptyset;
\]

\[
\text{partition}(r, C, r_1, \ldots, r_n);
\]

\[
\text{partition}(s, C, s_1, \ldots, s_n);
\]

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do}
\]

\[
\text{outer\_bucket} \leftarrow \text{read\_partition}(r_i);
\]

\[
\text{foreach page } p \in s_i \text{ do}
\]

\[
p \leftarrow \text{read\_page}(s_i);
\]

\[
\text{foreach tuple } x \in \text{outer\_bucket} \text{ do}
\]

\[
\text{foreach tuple } y \in p \text{ do}
\]

\[
\text{if } x[C] = y[C] \text{ and } \text{intsct}(x[T], y[T]) \neq \emptyset \text{ then}
\]

\[
\begin{align*}
  z[A] & \leftarrow x[A]; \\
  z[B] & \leftarrow y[B]; \\
  z[C] & \leftarrow x[C]; \\
  z[T] & \leftarrow \text{intsct}(x[T], y[T]); \\
  \text{result} & \leftarrow \text{result} \cup \{z\};
\end{align*}
\]

\[
\text{return result;}
\]
How to Partition Intervals?

- Given is a set of intervals (for a specific key). How shall they be partitioned?

- Intervals in one partition shall be as similar as possible
- Partition intervals according to **position and duration**

- Anton Dignös, Michael H. Böhlen, Johann Gamper: Overlap interval partition join. SIGMOD Conference 2014: 1459-1470
Partition-Based Join/3

**Algo:** partition\((r, C, r_1, \ldots, r_n)\)

```
for i ← 1 to n do
    \(r_i \leftarrow \emptyset;\)

foreach block \(b \in r\) do
    read_block\((b)\);
    foreach tuple \(x \in b\) do
        \(i \leftarrow \text{hash}(x[C]);\)
        \(r_i \leftarrow r_i \cup \{x\};\)

return result;
```
Index-based Join

- **Index-based join**: In the nested loop join, index lookups can replace scan of the inner relation if
  - join is an equi-join or natural join and
  - an index is available on the inner relation’s join attribute
    - Can construct an index just to compute a join.

- For each tuple \( x \) in the outer relation \( r \), use the index to look up tuples in \( s \) that satisfy the join condition.

- \( \text{Cost} = |r| \times c + |b_r| \)
  - where \( c \) is the cost of traversing the index and fetching all matching \( s \)-tuples for one tuple of \( r \)

- If indexes are available on join attributes of both \( r \) and \( s \), use relation with fewer tuples as the outer relation.
Temporal aggregation is substantially more complex than nontemporal aggregation.

Temporal aggregation is a good example to illustrate that the temporal support of nontemporal database is inadequate.

- small-scale aggregation: aggregation tree, balanced tree
- large-scale aggregation: bucket algorithm
Three main paradigms for join processing
- nested loop
- sort merge
- partitioning (hashing)

Mostly the techniques from nontemporal databases carry over to temporal databases directly.

Interval intersection must be added to traditional algorithms.

Temporal joins (and Cartesian products) add inequality conditions to the join condition.

Interval intersection usually leaves less room for optimization than equality on nontemporal attributes; always go first for optimization of nontemporal (equality) predicates.