

# Part4

## Temporal Joins and Temporal Aggregation

- ▶ Temporal Cartesian Product and Temporal Joins
  - ▶ Definition
  - ▶ Join types
  - ▶ Evaluation algorithms
- ▶ Temporal Aggregation
  - ▶ Definition
  - ▶ Aggregation tree
  - ▶ Bucket algorithm

# Literature and Acknowledgments

- ▶ D. Gao, C.S. Jensen, R.T. Snodgrass, and M.D. Soo: Join operations in temporal databases. *VLDB Journal*, 14:2–29, 2005.
- ▶ B. Moon, I.F. Vega Lopez, and V. Immanuel: Efficient algorithms for large-scale temporal aggregation. *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, 2004.
- ▶ These slides are based on slides from
  - ▶ Johann Gamper, Free University of Bozen-Bolzano, Italy

# Temporal Join Processing

- ▶ Query processing
- ▶ Definition of Temporal Cartesian
- ▶ Definition of Temporal Joins
  - ▶ Theta join
  - ▶ Equijoin
  - ▶ Natural join
- ▶ Evaluation algorithms for temporal joins
  - ▶ Nested-loop join
  - ▶ Sort-merge-based join
  - ▶ Partition-based join

# 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

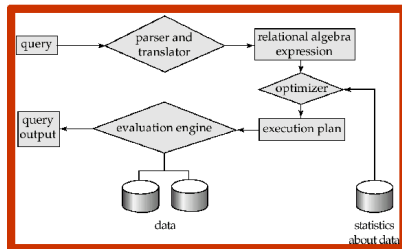
- ▶ Check syntax and verify relations.
- ▶ Translate the query into an equivalent relational algebra expression.

## 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**

# 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

# Notation

- ▶ Outset: Linear and discrete timeline partitioned into minimal-duration intervals (chronons)
- ▶ Two valid-time relations

$$\begin{aligned} R &= (A_1, \dots, A_n, T_s, T_e) \\ S &= (B_1, \dots, B_m, T_s, T_e) \end{aligned}$$

- ▶  $A_i, B_i$  are the explicit attributes
- ▶  $T_s, T_e$  are the timestamp start and end attributes
- ▶ Shorthand notations:  $T = [T_s, T_e]$ ,  $A = \{A_1, \dots, A_n\}$ ,  $B = \{B_1, \dots, B_n\}$
- ▶  $r, s$  are instances of  $R, S$

# Example

## ► Example:

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

Employee

| EmpName | Dept | T      |
|---------|------|--------|
| Ron     | Ship | [1,5]  |
| George  | Ship | [5,9]  |
| Ron     | Mail | [6,10] |

Manager

| Dept | MgrName | T      |
|------|---------|--------|
| Load | Ed      | [3,8]  |
| Ship | Jim     | [7,15] |

# Temporal Cartesian Product

- ▶ **Temporal Cartesian product** is a conventional Cartesian product that is
  - ▶ applied independently at each point in time, and
  - ▶ requires an intersection of the timestamp attributes.
- ▶ Auxiliary function on intervals
  - ▶ Function  $intsct(U, V)$ : returns the maximum interval contained in its two argument intervals and  $\emptyset$  if the intersection is empty.

$$intsct(U, V) = \begin{cases} [last(U_s, V_s), first(U_e, V_e)] & \text{if } last(U_s, V_s) \leq first(U_e, V_e) \\ \emptyset & \text{otherwise} \end{cases}$$

- ▶ **Definition:** Temporal Cartesian product

$$r \times^T s = \{z^{(n+m+2)} \mid \exists x \in r \exists y \in s (z[A] = x[A] \wedge z[B] = y[B] \wedge z[T] = intsct(x[T], y[T]) \wedge z[T] \neq \emptyset)\}$$

## Example: Temporal Cartesian Product

Employee

| EmpName | Dept | T      |
|---------|------|--------|
| Ron     | Ship | [1,5]  |
| George  | Ship | [5,9]  |
| Ron     | Mail | [6,10] |

Manager

| Dept | MgrName | T      |
|------|---------|--------|
| Load | Ed      | [3,8]  |
| Ship | Jim     | [7,15] |

Employee  $\times^T$  Manager

| EmpName | Dept | Dept | MgrName | T      |
|---------|------|------|---------|--------|
| Ron     | Ship | Load | Ed      | [3,5]  |
| George  | Ship | Load | Ed      | [5,8]  |
| George  | Ship | Ship | Jim     | [7,9]  |
| Ron     | Mail | Load | Ed      | [6,8]  |
| Ron     | Mail | Ship | Jim     | [7,10] |

# Temporal Theta Join and Equijoin

- **Definition:** Temporal theta join

$$r \bowtie_P^T s = \sigma_{P(r[A],s[B])}(r \times^T s)$$

- $P$  is a predicate over the nontemporal attributes in  $r$  and  $s$

- **Definition:** Temporal equijoin on attributes  $A' \subseteq A$  and  $B' \subseteq B$  is defined as the theta join with predicate  $P \equiv r[A'] = s[B']$ , i.e.,

$$r \bowtie_{r[A']=s[B']}^T s$$

# Temporal Natural Join

- Explicit join attributes:  $C = \{C_1, \dots, C_k\}$

$$R = (A_1, \dots, A_n, C_1, \dots, C_k, T_s, T_e)$$

$$S = (B_1, \dots, B_m, C_1, \dots, C_k, T_s, T_e)$$

- Result tuples have to agree on the join attributes

- **Definition:** Temporal natural join

$$r \bowtie^T s = \{z^{(n+m+k+2)} \mid \exists x \in r \exists y \in s (x[C] = y[C] \wedge \\ z[A] = x[A] \wedge z[B] = y[B] \wedge z[C] = y[C] \wedge \\ z[T] = \text{intsct}(x[T], y[T]) \wedge z[T] \neq \emptyset)\}$$

# 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

**Algorithm:** 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**

**if**  $x[C] = y[C]$  and  $intsct(x[T], y[T]) \neq \emptyset$  **then**

$z[A] \leftarrow x[A]$ ;  $z[B] \leftarrow y[B]$ ;  $z[C] \leftarrow x[C]$ ;

$z[T] \leftarrow intsct(x[T], y[T])$ ;

$result \leftarrow result \cup \{z\}$ ;

**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.

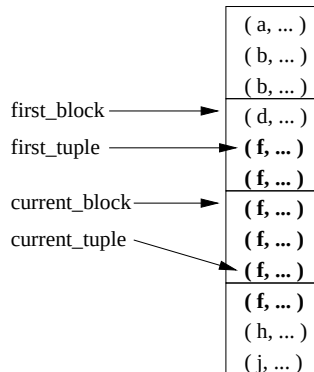
# Sort-Merge-Based Join/2

## ► Data structure to store **state information**

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



# Sort-Merge-Based Join/3

- Initialize the state of a scan

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

`state.current_block`  $\leftarrow 1$ ;

`state.current_tuple`  $\leftarrow 0$ ;

`state.first_block`  $\leftarrow \perp$ ;

`state.first_tuple`  $\leftarrow \perp$ ;

`seek(relation, state.current_block)`;

`state.tuples`  $\leftarrow \text{read\_block}(\text{relation})$ ;

## 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

**Algorithm:** `advance(relation, state)`

```
if state.current_tuple = MAX_TUPLES then
    state.tuples  $\leftarrow$  read_block(relation);
    state.current_block  $\leftarrow$  state.current_block + 1;
    state.current_tuple  $\leftarrow$  1;
else
    state.current_tuple  $\leftarrow$  state.current_tuple + 1;
```

- ▶ Return the next tuple in the scan

**Algorithm:** `currentTuple(state)`

**return** *state.tuple*[*state.current\_tuple*];

## Sort-Merge-Based Join/5

- Revert the current block and tuple counters to their last values

**Algorithm:** `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

**Algorithm:** `markScanStart(state)`

```
state.first_block  $\leftarrow$  state.current_block;
state.first_tuple  $\leftarrow$  state.current_tuple;
```

# Sort-Merge-Based Join/6

**Algorithm:** SortMergeJoin( $r, s, C$ )

```
 $r' \leftarrow \text{sort}(r, C); \quad s' \leftarrow \text{sort}(s, C);$   
 $\text{initState}(r', \text{outer\_state}); \quad \text{initState}(s', \text{inner\_state});$   
 $x'[C] \leftarrow \perp; \quad \text{result} \leftarrow \emptyset;$   
 $\text{advance}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_tuple}(\text{inner\_state});$   
for  $i \leftarrow 1$  to  $|r'|$  do  
   $\text{advance}(r', \text{outer\_state}); \quad x \leftarrow \text{current\_tuple}(\text{outer\_state});$   
  if  $x[C] = x'[C]$  then  
     $\text{backUp}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_tuple}(s', \text{inner\_state});$   
   $x'[C] \leftarrow x[C];$   
  while  $x[C] > y[C]$  do  
     $\text{advance}(s', \text{inner\_state}); \quad y \leftarrow \text{current\_tuple}(\text{inner\_state});$   
   $\text{markScanStart}(\text{inner\_state});$   
  while  $x[C] = y[C]$  do  
    if  $\text{intsct}(x[T], y[T]) \neq \emptyset$  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/1

- ▶ **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

## Partition-Based Join/2

**Algorithm:** PartitionJoin( $r, s, C$ )

```
result  $\leftarrow \emptyset$ ;  
partition( $r, C, r_1, \dots, r_n$ );  
partition( $s, C, s_1, \dots, s_n$ );  
for  $i \leftarrow 1$  to  $n$  do  
    outer_bucket  $\leftarrow$  read_partition( $r_i$ );  
    foreach page  $p \in s_i$  do  
         $p \leftarrow$  read_page( $s_i$ );  
        foreach tuple  $x \in$  outer_bucket do  
            foreach tuple  $y \in p$  do  
                if  $x[C] = y[C]$  and  $intsct(x[T], y[T]) \neq \emptyset$  then  
                     $z[A] \leftarrow x[A]$ ;  $z[B] \leftarrow y[B]$ ;  $z[C] \leftarrow x[C]$ ;  
                     $z[T] \leftarrow intsct(x[T], y[T])$ ;  
                    result  $\leftarrow$  result  $\cup \{z\}$ ;  
return result;
```

# Partition-Based Join/3

**Algorithm:** `partition( $r, C, r_1, \dots, r_n$ )`

**for**  $i \leftarrow 1$  *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.
- ▶  $Cost = |r| * c + |b_r|$ 
  - ▶ 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.

# Summary

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

# Temporal Aggregation

- ▶ Snapshot vs. temporal aggregation
- ▶ Aggregation tree algorithm
- ▶ Balanced tree algorithm
- ▶ Merge-sort algorithm
- ▶ Bucket algorithm

# Snapshot Aggregation

- ▶ **Snapshot aggregation** transforms an argument relation into a **summary result** relation
  - ▶ Result relation can be a single tuple (scalar aggregate: a single aggregation group) or several tuples (aggregate function: 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 all the employees.

```
SELECT avg(Salary)
FROM   Employee
```
  - ▶ Compute the average salary of employees grouped by department.

```
SELECT  Dept, avg(Salary)
FROM    Employee
GROUP BY Dept
```

# Temporal Aggregation

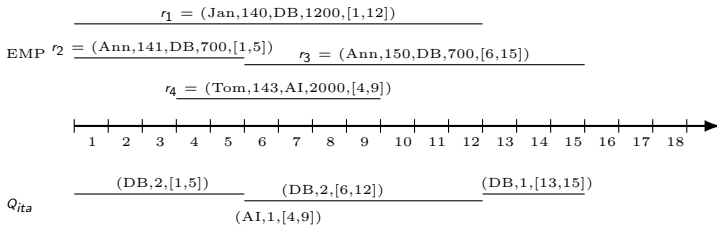
- ▶ Additionally use time dimension to group argument tuples.
- ▶ **Temporal grouping:** Process where the timeline is partitioned over time and argument tuples are grouped over these time partitions.
  - ▶ The aggregate values are computed over each of these groups in turn.
- ▶ Two types of temporal grouping:
  - ▶ **Span grouping:** The timeline is partitioned into fixed-length intervals, e.g., month or year.
    - ▶ Independent of argument tuples
  - ▶ **Instant grouping:** The 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
  - ▶ Moving-window temporal aggregation
  - ▶ Span temporal aggregation

# Different Forms of Temporal Aggregation/1

## ► Instant temporal aggregation (ITA)

- Compute aggregate results for each time point  $t$  over all tuples that are valid at  $t$
- Coalesce consecutive tuples with identical aggregate values into constant intervals.
  - Lineage information might be considered or not

►  $Q_{ita}$ : For each month and department, what is the number of contracts?



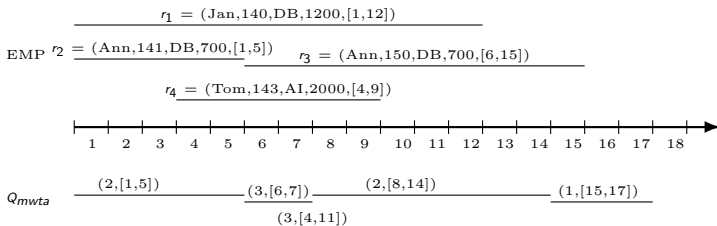
|       | Name | CID | Dept | Sal  | T       |
|-------|------|-----|------|------|---------|
| $r_1$ | Jan  | 140 | DB   | 1200 | [1, 12] |
| $r_2$ | Ann  | 141 | DB   | 700  | [1, 5]  |
| $r_3$ | Ann  | 150 | DB   | 700  | [6, 15] |
| $r_4$ | Tom  | 143 | AI   | 2000 | [4, 9]  |

| Dept | Cnt | T        |
|------|-----|----------|
| DB   | 2   | [1, 5]   |
| DB   | 2   | [6, 12]  |
| DB   | 1   | [13, 15] |
| AI   | 1   | [4, 9]   |

# Different Forms of Temporal Aggregation/2

## ► Moving-window temporal aggregation (MWTa)

- Compute aggregate results for each time point  $t$  over all tuples that are valid in  $[t - w, t]$ .
- Coalesce consecutive tuples as in ITA
- $Q_{mwt a}$ : *For each month, how many contracts have been in effect during this month and the preceding two months?*

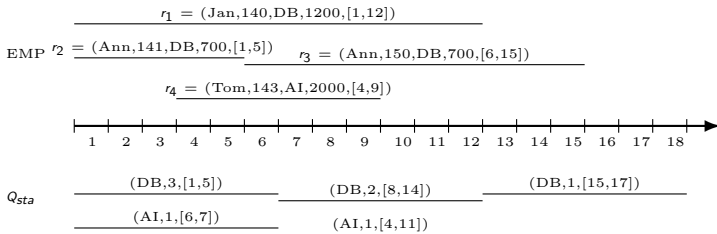


|       | Name | CID | Dept | Sal  | T      |
|-------|------|-----|------|------|--------|
| $r_1$ | Jan  | 140 | DB   | 1200 | [1,12] |
| $r_2$ | Ann  | 141 | DB   | 700  | [1,5]  |
| $r_3$ | Ann  | 150 | DB   | 700  | [6,15] |
| $r_4$ | Tom  | 143 | AI   | 2000 | [4,9]  |

| Dept | Cnt | T       |
|------|-----|---------|
| DB   | 2   | [1,5]   |
| DB   | 3   | [6,7]   |
| DB   | 2   | [8,14]  |
| DB   | 1   | [15,17] |
| AI   | 1   | [4,11]  |

# Different Forms of Temporal Aggregation/3

- **Span temporal aggregation (STA)**
  - Compute the aggregate results for each user-specified time period.
- $Q_{sta}$ : For each half-year period and department, what is the number of contracts?



|       | Name | CID | Dept | Sal  | T      |
|-------|------|-----|------|------|--------|
| $r_1$ | Jan  | 140 | DB   | 1200 | [1,12] |
| $r_2$ | Ann  | 141 | DB   | 700  | [1,5]  |
| $r_3$ | Ann  | 150 | DB   | 700  | [6,15] |
| $r_4$ | Tom  | 143 | AI   | 2000 | [4,9]  |

| Dept | Cnt | T       |
|------|-----|---------|
| DB   | 3   | [1,6]   |
| DB   | 2   | [7,12]  |
| DB   | 1   | [13,18] |
| AI   | 1   | [1,6]   |
| AI   | 1   | [7,12]  |
| AI   | 0   | [13,18] |

# Tuma's Work on Temporal Aggregation

- ▶ Epstein (1979) proposed a simple algorithm for the evaluation of nontemporal scalar aggregates.
- ▶ Tuma (1992) extended Epstein's work and proposed a two-step evaluation process for temporal instant aggregation:
  - ▶ **Step 1:** Compute the constant intervals
  - ▶ **Step 2:** Compute the aggregate functions
- ▶ The steps are separate and Step 1 must be completed before Step 2.
- ▶ Requires **two scans** of the argument relation!

# Aggregation Tree Algorithm

- ▶ 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.

# Aggregation Tree Algorithm - Incremental Construction/1

- ▶ 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

# Aggregation Tree Algorithm - Incremental Construction/2

- ▶ **Example:** Employee relation of a company

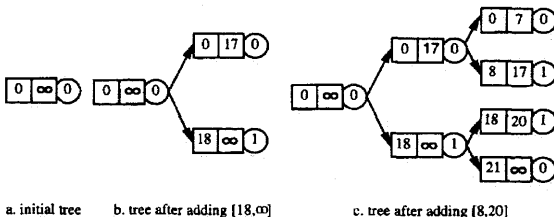
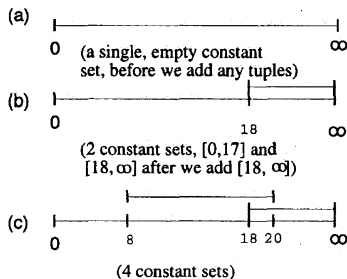
Employee

| Name    | Salary | Start | End      |
|---------|--------|-------|----------|
| Richard | 40K    | 18    | $\infty$ |
| Karen   | 45K    | 8     | 20       |
| Nathan  | 35K    | 7     | 12       |
| Nathan  | 37K    | 18    | 21       |

- ▶ Assume a time domain from 0 to  $\infty$ .
- ▶ Query: *Compute the number of tuples valid over each constant interval*

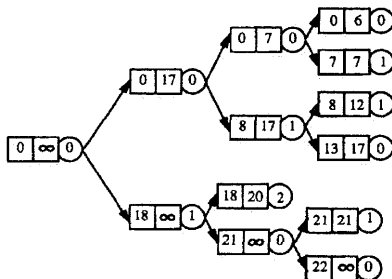
# Aggregation Tree Algorithm - Incremental Construction/3

- Initially, a single node valid from 0 to  $\infty$  has a count of 0
- Add the tuple (*Richard*, 40K, 18,  $\infty$ ): The start time 18 requires to split the root node
- Add the tuple (*Karen*, 45K, 8, 20): The start time and the end time require a node split



# Aggregation Tree Algorithm - Incremental Construction/4

d. Add the tuple (*Nathan*, 35K, 7, 12) and (*Nathan*, 37K, 18, 21)



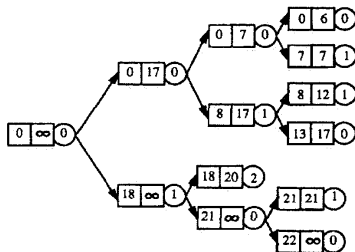
d. final tree after adding [7,12] and [18,21]

# Aggregation Tree Algorithm - Producing 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:  $0 + 0 + 1 + 1 = 2$



d. final tree after adding  $[7, 12]$  and  $[18, 21]$

Result

| Count | Start | End      |
|-------|-------|----------|
| 0     | 0     | 6        |
| 1     | 7     | 7        |
| 2     | 8     | 12       |
| 1     | 13    | 17       |
| 3     | 18    | 20       |
| 2     | 21    | 21       |
| 1     | 22    | $\infty$ |

# Aggregation Tree Algorithm - Complexity

- ▶ Requires **only one scan** of the argument relation.
- ▶ Average complexity to construct the aggregation tree:  $\mathcal{O}(n \log n)$ , where  $n$  is the number of argument tuples
- ▶ Worst case complexity:  $\mathcal{O}(n^2)$ 
  - ▶ If input tuples are sorted in chronological order, the tree ends up in a linear list
- ▶ Scan or lookup of aggregate values in the aggregation tree is ( $\mathcal{O}(n)$ . (Negligible compared to construction cost.)
- ▶ Goof if the aggregation tree fits into memory (when timestamps are duplicated).

# Balanced Tree Algorithm/1

- ▶ The **balanced tree algorithm** is built on the following two ideas:
  - ▶ Timestamp sorting (basic idea)
  - ▶ Creating a balanced tree (optimization; 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.

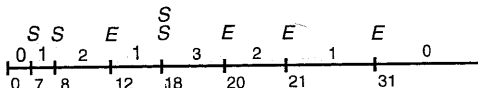
# Balanced Tree Algorithm/2

- **Example:** Compute the time-varying number of tuples for the following Employee relation.

| Employee |        |            |       |     |
|----------|--------|------------|-------|-----|
| Name     | Salary | Dept       | Begin | End |
| Richard  | 46,000 | Accounting | 18    | 31  |
| Karen    | 45,000 | Shipping   | 8     | 20  |
| Nathan   | 35,000 | Marketing  | 7     | 12  |
| Nathan   | 38,000 | Accounting | 18    | 21  |

| Result |       |     |
|--------|-------|-----|
| Count  | Begin | End |
| 1      | 7     | 7   |
| 2      | 8     | 11  |
| 1      | 12    | 17  |
| 3      | 18    | 19  |
| 2      | 20    | 20  |
| 1      | 21    | 31  |

- Timestamp sorting to compute the count aggregate function



## Balanced Tree Algorithm/3

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

**Algorithm:** `BalancedTree( $r$ )`

$\mathcal{T} \leftarrow$  empty balanced tree;

**foreach** *tuple*  $t \in r$  **do**

**if**  $t.start\_time = n.ts$  for any node  $n$  in  $\mathcal{T}$  **then**

$n.no\_starts++$ ;

**else**

        Insert a new node  $n'$  with  $n'.ts = t.start\_time$  into  $\mathcal{T}$ ;  $n'.no\_starts \leftarrow 1$ ;

**if**  $t.end\_time = n.ts$  for any node  $n$  in  $\mathcal{T}$  **then**

$n.no\_ends++$ ;

**else**

        Insert a new node  $n'$  with  $n'.ts = t.end\_time$  into  $\mathcal{T}$ ;  $n'.no\_ends \leftarrow 1$ ;

$count \leftarrow 0$ ;

**foreach** *node*  $n \in \mathcal{T}$  *traversed by in-order* **do**

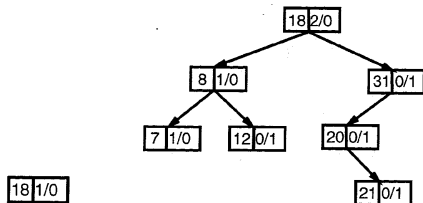
$count \leftarrow count + n.no\_starts$ ;

$count \leftarrow count - n.no\_ends$ ;

    Output  $n.ts$  and  $count$ ;

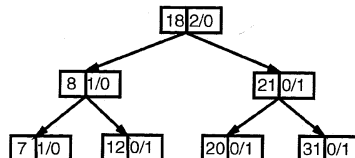
# Balanced Tree Algorithm/4

- **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.



(a)

(b)



(c)

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

# Balanced Tree Algorithm - Complexity and Limitations

## ► Complexity

- Requires only **one scan** of the relation
- $\mathcal{O}(n \log n)$  (worst case)

## ► Limitations

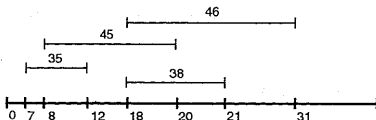
- Works also for sum and avg aggregation function.
  - For the sum function store at each node the sum of the attribute values of the starting/ending tuples
  - For avg function store the pair (sum, count)
- **Not applicable** for min and max function
  - The balanced tree does not keep track of the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

# Merge-Sort Algorithm/1

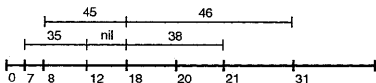
- ▶ **Merge-sort algorithm** is a bottom-up aggregation approach for `min` and `max` functions
  - ▶ Computes a larger (intermediate) aggregate result by merging two smaller (intermediate) aggregate results
- ▶ Intermediate aggregate is defined as a pair  $(T_k, M_k)$  where
  - ▶  $T_k = \{t_0, \dots, t_k\}$  is a set of unique timestamps ( $t_0 < \dots < t_k$ )
  - ▶  $M_k = \{m_0, \dots, m_k\}$  is a set of attribute values, where  $m_i$  is a maximum/minimum attribute value associated with a time interval  $[t_{i-1}, t_i)$ .
- ▶ **Algorithm**
  - ▶ The first scan of the relation generates  $\lceil n/2 \rceil$  intermediate aggregates
  - ▶ Then the algorithm recursively merges the intermediate results until a final aggregate result is obtained
- ▶  $\mathcal{O}(n \log n)$  complexity

# Merge-Sort Algorithm/2

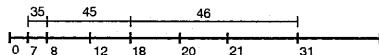
- ▶ **Example:** Compute the maximum salary using the merge-sort aggregation algorithm.
- ▶ Initial relation



- ▶ First merging pass: merge tuples 1 and 2 and tuples 3 and 4



- ▶ Second merging pass: merge the two intermediate results to get the final result

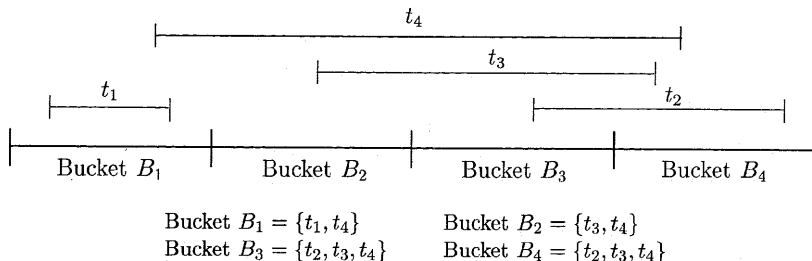


# Bucket Algorithm

- ▶ 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

# Bucket Algorithm - Assign Tuples to Buckets/1

- **First approach:** Assign a tuple to all buckets it overlaps

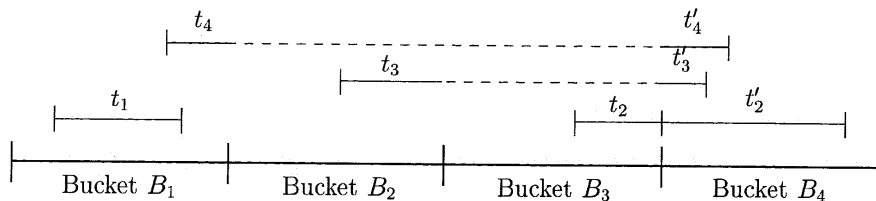


- Leads to considerable duplication of data, especially for long-lived tuples.

# Bucket Algorithm - Assign Tuples to Buckets/2

## ► Second approach:

- Suppose that the life span of a tuple  $t$  overlaps buckets  $B_i, B_{i+1}, \dots, B_j$
- 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.,  $B_{i+1}, \dots, B_{j-1}$ .
  - The size of the meta array is equal to the total number of buckets



Bucket  $B_1 = \{t_1, t_4\}$

Bucket  $B_2 = \{t_3\}$

Bucket  $B_3 = \{t_2\}$

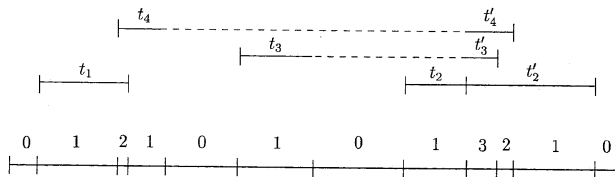
Bucket  $B_4 = \{t'_2, t'_3, t'_4\}$

Meta Array for count aggregation =  $\{0, 1, 2, 0\}$

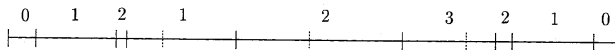
# Bucket Algorithm - Computing aggregates/3

## ► 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.



(a)



(b)

# Bucket Algorithm - Computing aggregates/4

**Algorithm:** BucketAlgorithm( $r$ )

$\mathcal{I}_B \leftarrow$  time interval for each bucket  $((T_{max} - T_{min})/\mathcal{N}_B)$ ;

**foreach** tuple  $t \in r$  **do**

$start\_bucket \leftarrow (t.start\_time - T_{min})/\mathcal{I}_B$ ;

$end\_bucket \leftarrow (t.end\_time - T_{min})/\mathcal{I}_B$ ;

    Insert  $t$  into a bucket  $B_{start\_bucket}$ ;

**if**  $start\_bucket \neq end\_bucket$  **then**

        └ Insert  $t'$  into a bucket  $B_{end\_bucket}$ ;

**for**  $i = start\_bucket + 1$  **to**  $end\_bucket - 1$  **do**

        └ Update  $meta\_array[i]$ ;

**for**  $i = 0$  **to**  $\mathcal{N}_B - 1$  **do**

    Perform temporal aggregation on the bucket  $B_i$ ;

    Combine the scalar value of  $meta\_array[i]$  to the bucket  $B[i]$ ;

    Merge the bucket boundary with  $B_{i-1}$  as needed;

# Bucket Algorithm - Complexity

- ▶ 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

# Summary

- ▶ Three join processing paradigms: nested loop, sort merge, partitioning
- ▶ Index lookups can replace the scan of the inner relation in nested loop joins.
- ▶ Temporal joins (and Cartesian products) add inequality conditions to the join condition.
- ▶ Inequality conditions are harder to optimize than equality conditions (partitioning not applicable; inequalities tend to be less selective than equalities; comparison is more difficult than equality).
- ▶ Temporal aggregation is substantially more complex than nontemporal aggregation.
- ▶ small-scale aggregation: aggregation tree, balanced tree, merge-sort
- ▶ large-scale aggregation: bucket algorithm