HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Join

Prof. Hasso Plattner

What is a Join?

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Combining tuples from different tables

- General categories:
  - Inner Join – Combine the tuples of two tables by combining each tuple of the first input table with each tuple of the second table to apply a join-predicate, which joins two tuples only if they match
  - Outer Join – If the join predicate matches, tuples from both join relations are used, if not, the non-matching tuple is filled with NULL values
- Further Specializations
  - Equi-Join – satisfying an equality predicate
  - Semi-Join – only one half of the join result is actually used

Joins in Applications

- Typically used in normalized databases to combine information from multiple tables
- Modeled as foreign keys
  - 1:1 – for each tuple on the left side one matching tuple on the right side exists
  - 1:n – for each tuple on the left side *n* tuples on the right side exist
  - n:m - for each tuple on the left side *n* tuples on the right exist and vice versa (*modeled using intermediate tables)*

Join Execution in Main Memory

- Leverage sequential scans
- Avoid random lookups if possible
- Avoid early materialization, work mostly with positions

Join Algorithms:

- Hash Join:
  - Build hash map for smaller join relation
  - Sequential scan over larger join relation and probe into hash map
- Sort-Merge Join:
  - Principle of merging two sorted lists
  - Less exception handling compared to hashing

Join Example

What is the **population density** for the cities of **Hessen**?

→ Extract information from two tables: world_population and locations

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

Assumption for this example: city names are unique

Hash Join

- Requirements: hash function that allows to map variable length values to fixed length keys, e.g. mapping the city name to a 64-bit unsigned integer value
- Hash join algorithm – two phases:
  1. Hash phase: during the hash phase the join attribute of the first join table is scanned and a hash map is produced where each value maps to the position of this value in the first table
  2. Probe phase: during the probe phase the second join table is scanned on the other join attribute and for each tuple the hash map is probed to determine if the value is contained or not; if the value is contained the result is written

Hash Join Example

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

world_population

recID	fname	lname	city	...
1	John	Doe	Berlin	...
2	Manfred	Mueller	Berlin	...
...	...	...	...	...
23	Max	Mustermann	Kassel	...
24	Hans	Gerhardt	Fulda	...

dictionary of city in wp

valueID	city
0	Aachen
1	Berlin
...	...
471	Kassel

locations

recID	city	state	country	area
1	Berlin	Berlin	Germany	851
2	Potsdam	Brandenburg	Germany	187
...	...	...	...	
902	Kassel	Hessen	Germany	107

Hash Join Execution (1)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

Positions from **locations** table: `CitiesInHessen = <902,912,1023,4581,…>`

Find all predicates that can be evaluated before the join execution starts

→ Filter **locations** table for state **"Hessen"** and return matched positions

Hash Join Execution (2)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

Hash map created in hash phase

hash value	positions in locations
hash(81)	902
hash(37)	912
hash(42)	1023
…	…

- Hash phase: Build hash map for cities of Hessen
- Scan *locations* table and map each city's valueID to the positions (i.e. in this case one position) of it in the table

Hash Join Execution (3)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

valueID mapping

value	*wp* valueID	*locations* valueID
Darmstadt	201	37
Fulda	300	42
Kassel	471	81
…	…	…

dictionary of city in *wp*

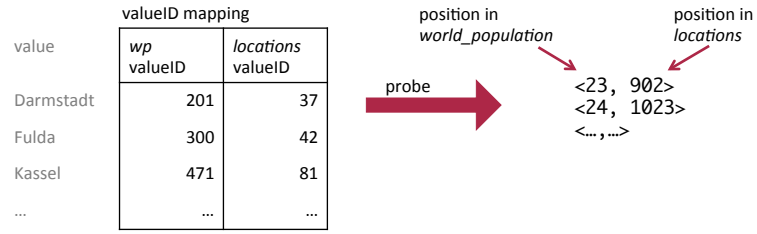
valueID	city
0	Aachen
1	Berlin
…	…
471	Kassel

1M cities

- Build mapping structure based on positions of the locations table (valueID in *wp* → valueID in *locations*)
- Lookup all encoded IDs from the dictionary of the joined table

Hash Join Execution (4)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```



- Scan world_population.city and probe each value in the hash map
- If the value is found emit position pair

Hash Join Execution (5)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

```
<23, 902>
<24, 1023>
<…,…>
```

- In the last phase of the query execution we need to materialize and aggregate the result
- For each tuple in the result list, materialize the value of the location from the world_population table and the name of the major from the locations table:

population	wp.city	locations.area
195530	Kassel	107
64349	Fulda	104
	...	...

Sort-Merge Join

- Hashing introduces additional complexity because we cannot guarantee the absence of collisions
- Typically slower than hash join, but easier to implement

Algorithm:

1. Filter **locations** table for all matching cities
2. Create list of cities and their valueIDs and a unique list of valueIDs for **locations**
3. Build mapping structure based on sorted (2) and mapped values in **world_population**
4. Create list similar to (2) for **world_population**
5. Sort and merge both lists to join

Sort-Merge Join Example

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

world_population

recID	fname	lname	city	...
1	John	Doe	Berlin	...
2	Manfred	Mueller	Berlin	...
...	...	...	...	...
23	Max	Mustermann	Kassel	...
24	Hans	Gerhardt	Fulda	...

locations

recID	city	state	country	area
1	Berlin	Berlin	Germany	851
2	Potsdam	Brandenburg	Germany	187
...	...	...	...	...
902	Kassel	Hessen	Germany	107

dictionary of city in wp

valueID	city
0	Aachen
1	Berlin
...	...
471	Kassel

dictionary of city in locations

valueID	city
0	Aachen
1	Berlin
...	...
81	Kassel

Sort-Merge Join (1)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

Find all predicates that can be evaluated before the join execution starts

→ Filter **locations** table for state **"Hessen"** and return matched positions

Sort-Merge Join (2)

```
SELECT wp.city , (COUNT(*) / locations.area) AS
population_density FROM world_population AS wp
INNER JOIN locations ON wp.city = locations.city
WHERE locations.state = 'Hessen'
GROUP BY wp.city , locations.area
```

Positions from **locations** table

```
CitiesInHessen =
<902,912,1023,…>
```

city column of locations

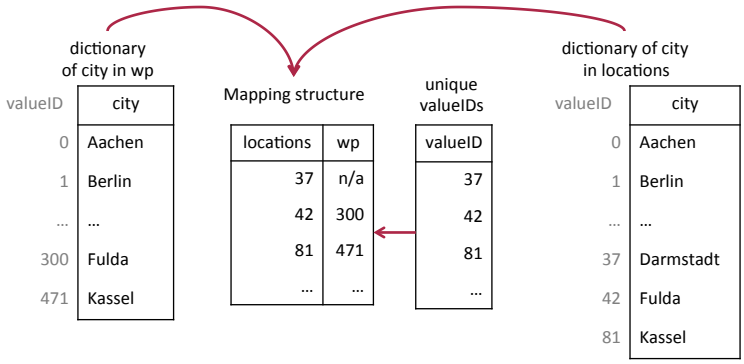
value	recID	valueID
Kassel	902	81
Darmstadt	912	37
Fulda	1023	42
…	…	

dictionary of city in locations

valueID	city
0	Aachen
1	Berlin
…	…
81	Kassel

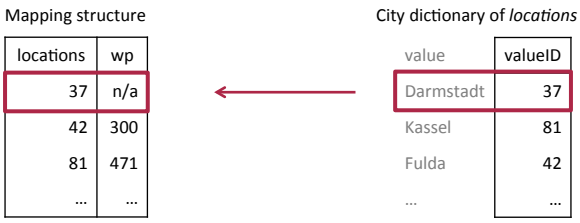
- Build a list of all occurrences of the predicated positions containing pairs of position and encoded value **L<pos, valueID>** and afterwards sort list by valueID
- Build a unique list **L<valueID>** containing all distinct encoded IDs

Sort-Merge Join (3)



- Create mapping structure for the join attributes of both relations
- Let us assume there is no population for Darmstadt and the mapping structure will thus not have a mapping for *valueID* 37

Sort-Merge Join (4)



- Remove all values from the mapping that have no join partner in the list of unique encoded IDs

Sort-Merge Join (5)

New list L<recID, valueID$_{locations}$, valueID$_{wp}$> for *locations* table

value	recID	old valueID	new valueID
Fulda	1023	42	300
~~Darmstadt~~	~~912~~	~~37~~	-
Kassel	902	81	417
...	...	...	...

Mapping structure

locations	wp
42	300
81	471
...	...

- Extend list for *locations* by adding corresponding valueIDs of *world_population* using the mapping structure
- Items that do not have a matching valueID in the mapping structure are removed (in our example *Darmstadt)*

Sort-Merge Join (6)

city column of locations

value	recID	old valueID	new valueID
Fulda	1023	42	300
Kassel	902	81	471
...	...	...	...

city column of wp

value	recID	valueID
Fulda	6002	300
Kassel	4711	471
Kassel	5039	471
...	...	...

table	position	valueID	valueID	position	table
locations	902	81	471	4711	wp
locations	902	81	471	5039	wp
locations	1023	42	300	6002	wp
...	...	...	...	...	...

- Combine results from both lists by merging them sequentially

Which Join Algorithms to choose?

- When to choose which algorithm?
  - Nested-Loop Join – very small data set; both other algorithms would require too much time for additional data structures and sorting
  - Hash Join – index available on the attribute to make it easier to retrieve the hash map for the hash phase, one relation significantly smaller
  - Sort-Merge Join – join column(s) already sorted, default fallback
- Complexity comparison
  - Nested-Loop Join – O(N*M)
  - Hash Join – O(N+M)
  - Sort-Merge Join – O(N*log(N) + M*log(M))
- But:
  - Hash Join performs best if the hash map is expected to be very small and probably cache resident
  - Sort-merge Join on already sorted input has a complexity of O(N+M) since the data does not need to be sorted

Sort-Merge Join (5)

Old L<recID, valueID> for *locations*

value	recID	old valueID
Kassel	902	81
Darmstadt	912	37
Fulda	1023	42
…	…	…

New L<recID, valueID> for *locations*

value	recID	new valueID
Kassel	902	471
~~Darmstadt~~	-	-
Fulda	1023	300
…	…	…

Mapping structure

wp	location
300	42
471	81

- Adapt list for *locations* by changing valueIDs to match valueIDs of *world_population* using the mapping structure
- Items that do not have a matching valueID in the mapping structure are removed (in our example *Darmstadt)*