Join Operator

Algorithms Preface:

**A1 – Nested Loop**
- Takes \( rIDa, rIDb, \text{key} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**A2 – Block Nested Loop**
- Takes \( rIDa, rIDb, \text{key}, \# \text{of blocks} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**A3 – Indexed Nested-Loop**
- Takes \( rIDa, rIDb, \text{key}, \# \text{of blocks} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**A4 – Merge**
- Takes \( rIDa, rIDb, \text{key}, \# \text{of blocks} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**A5 – Hybrid Merge**
- Takes \( rIDa, rIDb, \text{key}, \# \text{of blocks} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**A6 – Hash**
- Takes \( rIDa, rIDb, \text{key}, \# \text{of blocks} \)
- Returns \( \text{Iterator<tuple>}, \text{status} \)

**API**

*No constructor because Join is an operation internal to the query processor.*

**evaluate(tuplea, tupleb)**

*Returns status good or bad*
- This is a member function (part of the predicate tree object) used to evaluate whether the given relations can satisfy the query.

**join(SI<tuple>a, SI<tuple>b, predicate tree, # of blocks, JoinType)**

*Returns sorted smarterator of tuples*
- The join function takes in two relations, \( rIDa \) and \( rIDb \), and based off the information provided in the predicate tree decides which join algorithm, A1-A7, is appropriate to use. It also takes information about the number of blocks free in the buffer pool (OS query) that factors into the algorithm selection. The JoinType specification defines the type of join being performed (inner, left, right, or full).

Depending on the arguments that the query processor passes to these functions, several optimization exists to compute the result set of the theta join operation.

If the join operator can determine that the predicate tree indicates a natural join or an equi-join operation, then the join operator can use the merge-join algorithm to compute the result set. In order to use the merge-join algorithm, the join operator needs to know the join attributes, so the join operator will have an internal function to get the join attributes from the predicate tree. Another requirement to use the merge-join algorithm is that the first two parameters both needs to be sorted by the join attributes. If the join operator cannot determine that either of the parameters have a primary index on the join attributes or that
either of the parameters are sorted by the join attributes, then the join operator will need to call the sort
operator on the unsorted parameters for each join attribute or use the hash-join algorithm instead.

If the join operator can determine that the predicate tree indicates a natural join or an equi-join operation,
that either the first or the second parameter is unsorted, and that all unsorted parameters have a secondary
B+-tree index on the join attributes, then the join operator can use the hybrid merge-join algorithm to
compute the result set. When using the merge-sort algorithm, the join operator will call the searchAll
function from the B+ tree API in order to get the key and tuple-id pairs, and the join operator will also
call the sort operator to sort on the tuple-ids.

If the join operator can determine that the predicate tree indicates a natural join or an equi-join operation,
that the first and second parameters are both unsorted, and that there are no indexes on the join attributes,
the join operator can use the hash-join algorithm to compute the result set. When using the hash-join
algorithm, the join operator will use the Extensible Dynamic Hashing Index API to create the partitions
for each relation and to create the in-memory hash index.