FLIP-229: Introduces Join Hint for Flink SQL Batch Job

Motivation

In SQL jobs, join is a very common computing operator. Currently, Flink can automatically select an appropriate join strategy for the SQL based on statistical information of the source table, and generate an execution plan to run the job. However, due to various reasons, such as missing or incorrect statistical information, the execution plan generated by the optimizer is not optimal, and the optimizer will use the wrong join strategy, resulting in a poor final optimization effort or even failing to execute.
The join hint is a common solution in the industry to improve the shortcomings of the optimizer by intervening in optimizing the plan. The appendix[1] lists the join hints supported by the current popular computing engines and databases. By introducing join hints, users can intervene in the selection of the join strategy in optimizer, and manually optimize the execution plan to improve the performance of the query.
Currently, join hint is a feature that many users urgently need and there has been a lot of discussion about it in the community. For example,

1. https://issues.apache.org/jira/browse/FLINK-25596
   Use hint to specify Hash Join and Sort Merge Join
2. https://cwiki.apache.org/confluence/display/FLINK/FLIP-204%3A+Introduce+Hash+Lookup+Join
   Introduce a join hint named SHUFFLE_HASH in lookup join.
3. https://issues.apache.org/jira/browse/FLINK-20670
   Support to introduce join hints in SQL.

SQL Syntax

Although Flink does not support Join Hint in SQL now, the related SQL syntax has been proposed in Flip-113 [2] and Flip-204 [3], so it's unnecessary to repeat introducing that here.
The SQL syntax just like the following:

```
query:
  select /*+ hint_content[, hint_content] */ ...

hint_content:
```

```
  hint_strategy_name(hint_item[,hint_item])

hint_strategy_name:
  supported_join_hint_name

hint_item:
  string_literal
```

Support Strategies

The following Join strategies are currently supported in Flink SQL for batch job:

- Broadcast Join
  In this Join strategy, the data on the build side (usually a small table) will be broadcast to each downstream operator, and the data on the probe side (usually a large table) will be sent directly to the downstream operator with Forward. Then the data on the build side (small table) will be built into a Hash Table for the probe side to query.
- Hash Shuffle Join
  In this Join strategy, the data on the Build side and the Probe side will be shuffled with the join key, and the data with the same key will be distributed to the same downstream operator. Then the data on the build side (smaller table) will be built into a Hash Table for the probe side to query.

  Compared with the SHUFFLE strategy, the BROADCAST strategy does not need to shuffle the probe side, which saves a lot of shuffle time. Therefore, when a table is extremely small, the BROADCAST strategy is usually selected to avoid the shuffle cost and improve computing performance. However, when the scale of the small table is large, the BROADCAST strategy is not applicable, because the overhead of redundant data transmission will exceed the overhead of shuffle.
- Sort Merge Join
  This Join strategy is aimed at the scenario of joining between two large tables or the scenario that the data at both sides of the join is already in order. This strategy first shuffles the data on both sides of the join to the downstream operator according to the Join Key. Then the downstream operator sorts the data before joining, and finally joins the data at both ends. This strategy eliminates the need to load all data on one side into memory, thus relieving the pressure on computational memory.
- Nested Loop Join

In this Join strategy, the probe side is used as an Outer Loop, and the build side is used as an Inner Loop, and the data is joined through two-layer loops.
Currently, the two Join strategies, Broadcast and Shuffle, are only slightly different in distribution logic, so the same physical operator BatchPhysicalHashJoin is reused between these two strategies. The physical operator corresponding to the Sort Merge strategy is BatchPhysicalSortMergeJoin, and the physical operator corresponding to the Nested Loop strategy is BatchPhysicalNestedLoopJoin.
Based on the above, it is proposed that Flink supports the following Join Hint strategies: (case insensitive)

- BROADCAST
  Only supports join with equivalence except Full Outer Join.
  For example,

```
// t1 is the broadcast table
select /*+ BROADCAST(t1) */ * from t1
join t2 on t1.a = t2.a

// when join between t1 and t2, t1 will be chosen as the broadcast table
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the broadcast table
select /*+ BROADCAST(t1,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c

// BROADCAST does not support non-equivalent joins,
// so the planner's default strategy is adopted
select /*+ BROADCAST(t1) */ * from t1
join t2 on t1.a > t2.a

// BROADCAST does not support non-equivalent joins,
// so the planner's default strategy is adopted
select /*+ BROADCAST(t1) */ * from t1
full outer join t2 on t1.a = t2.a
```

- SHUFFLE_HASH
  Only supports join with equivalence.
  For example,

```
// t1 is the build side
select /*+ SHUFFLE_HASH(t1) */ * from t1
join t2 on t1.a = t2.a

// when join between t1 and t2, t1 will be chosen as the build side
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the build side
select /*+ SHUFFLE_HASH(t1,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c
```

```
// SHUFFLE_HASH does not support non-equivalent join,
// so the planner's default strategy is adopted
select /*+ SHUFFLE_HASH(t1) */ * from t1
join t2 on t1.a > t2.a
```

- SHUFFLE_MERGE
  Only supports join with equivalence.
  For example,

```
// Sort Merge Join is adopted
select /*+ SHUFFLE_MERGE(t1) */ * from t1
join t2 on t1.a = t2.a

// Sort Merge Join is both adopted in these two joins
select /*+ SHUFFLE_MERGE(t1,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c

// SHUFFLE_MERGE does not support non-equivalent join,
// so the planner's default strategy is adopted
select /*+ SHUFFLE_MERGE(t1) */ * from t1
join t2 on t1.a > t2.a
```

- NEST_LOOP
  Both support equivalent and non-equivalent joins.
  For example,

```
// t1 is the build side
select /*+ NEST_LOOP(t1) */ * from t1
join t2 on t1.a = t2.a

// when join between t1 and t2, t1 will be chosen as the build side
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the build side
select /*+ NEST_LOOP(t1,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c
```

Proposed Changes

Join Hint Strategy Definition

According to Flip-204 [3], we can directly define Join Hint through FlinkHintStrategies, taking SHUFFLE_HASH as an example:

```
HintStrategyTable
    .builder()
    .hintStrategy(
        "SHUFFLE_HASH",
        HintStrategy.builder(
            HintPredicates.or(
                HintPredicates.JOIN, HintPredicates.CORRELATE))
```

```
        .optionChecker(...)
        .build())
    .build();
```

We can put all the Join Hint strategies that we plan to support and check the number of parameters through the "optionChecker".

FlinkLogicalJoin Refactor

FlinkLogicalJoin needs to perceive Hint, so it needs to be refactored. Refer to Flip-204 [3] for more details.

Join Hint Validation

1. Join Hint syntax check

The syntax check contains the following,

- Unknown names of Join Hint
- Incorrect number of parameters

The logic of Hint syntax verification will be automatically completed by Calcite without modification, and the behavior when meets errors is to throw an exception.

2. Join Hint semantic check

The semantic check contains the following,

- The specified table name or view name does not exist

The behavior of semantic validation is before optimizing the SQL, and the behavior when meets errors is to throw an exception.

3. Check when optimizing

In the optimization stage of the optimizer, it is judged that this Join Hint can be applied. If it cannot be applied, the behavior is to print a warning log instead of throwing an exception directly.

Optimizer Rules

This Flip mainly involves in Flink SQL for batch job, so the main modified Rules are the following:

- BatchPhysicalHashJoinRule
- BatchPhysicalSortMergeJoinRule
- BatchPhysicalNestedLoopJoinRule
- BatchPhysicalSingleRowJoinRule

(For the support of Join Hint on Lookup Join, please refer to Flip-204[3])

In these rules, an util class will be abstracted to whether to adopt a certain Join strategy according to the actual join node and join hint.

This util class first determines whether the current join strategy can be applied to the join node. If the join node does not satisfy the Join strategy (for example, the shuffle hash join strategy cannot

be applied to non-equivalent joins), the strategy is skipped. When a certain join strategy can be applied to the current Join, it is further judged whether the join hint provided by the user matches the Join strategy. If it matches, the Hint is valid, and this Join strategy is adopted.
The logic above only needs to be added to the method of the current Rules named "matches". Part of the pseudocode of the util is as follows:

```
// check whether this join Strategy is valid. E.g, the join doesn't
// contain at least one equal-join condition or the table config
// forbids using SortMergeJoin.
if(!checkJoinStrategyValid(join, tableConfig, joinStrategy)){
  return false;
}

List hints = join.getHints;
List validHints = new ArrayList();

// collect the valid join hints
hints.forEach(hint ->{
  JoinStrategy strategy = JoinStrategy.getJoinStrategy(hint.hintName);
  if(checkJoinStrategyValid(join, tableConfig, strategy)){
    validHints.add(strategy);
  }
})

// if all join hints are invalid, treat it like no join hints
if(validHints.size() == 0){
  return true;
}

// if multiple hints are valid, pick the first one to apply
return validHints.sort().get(0) == joinStrategy;
```

Special Status With Hints

We refer to the behavior of join hint propagation and join hint conflict on popular computing engines and DBs. For details, please refer to Appendix [1]. We formulate the behavior of join hints in Flink for the following.

Hint propagation into view or subquery

Join Hint only affects the current query block, and does not affect the Join strategy in subquery and view. If the join hint really needs to be propagated into a view or subquery, we can discuss it in future.
For example,

```
// view
create view view1 as select t1.* from t1 join on t2 on t1.a = t2.a
```

```
// BROADCAST will not be propagated into view1,
// and view1 will use the planner's default join strategy
select /*+ BROADCAST(t1) */ * from view1 join t3 on view1.a = t2.a

// the join in view1 will use the planner's default join strategy,
// and the join between view1 and t1 will use BROADCAST
select /*+ BROADCAST(t1) */ * from view1 join t1 on view1.a = t1.a
```

Conflict in one same hints

If the hint declares the both sides of the join as the build side, the side being written first will finally be treated as the build side. For example,

```
// the first will be chosen, and t2 is the broadcast table
select /*+ BROADCAST(t2), BROADCAST(t1) */ * from t1
join t2 on t1.a = t2.a

// t1 is the broadcast table
select /*+ BROADCAST(t1,t1) */ * from t1
join t2 on t1.a = t2.a

// the first will be chosen, and t2 is the broadcast table
select /*+ BROADCAST(t2,t1) */ * from t1
join t2 on t1.a = t2.a

// the first will be chosen, and t2 is the broadcast table
select /*+ BROADCAST(t2,t1), BROADCAST(t1,t2) */ * from t1
join t2 on t1.a = t2.a

// just like BROADCAST(t1, t2) + BROADCAST(t3)
// when join between t1 and t2, t1 will be chosen as the broadcast table
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the broadcast table
select /*+ BROADCAST(t1,t2,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c

// just like SHUFFLE_HASH(t1, t2) + SHUFFLE_HASH(t3)
// when join between t1 and t2, t1 will be chosen as the build side
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the build side
select /*+ SHUFFLE_HASH(t1,t2,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c

// Sort Merge Join is adopted
select /*+ SHUFFLE_MERGE(t1,t1) */ * from t1
join t2 on t1.a = t2.a

// Sort Merge Join is adopted
select /*+ SHUFFLE_MERGE(t2), SHUFFLE_MERGE(t1) */ * from t1
join t2 on t1.a = t2.a

// Sort Merge Join is adopted
select /*+ SHUFFLE_MERGE(t2,t1) */ * from t1
join t2 on t1.a = t2.a
```

```
// Sort Merge Join is adopted
select /*+ SHUFFLE_MERGE(t2,t1), SHUFFLE_MERGE(t1,t2) */ * from t1
join t2 on t1.a = t2.a

// Sort Merge Join is both adopted in these two joins
select /*+ SHUFFLE_MERGE(t1,t2,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c

// the first will be chosen, and t1 is the build side
select /*+ NEST_LOOP(t2), NEST_LOOP(t1) */ * from t1
join t2 on t1.a = t2.a

// t1 is the build side
select /*+ NEST_LOOP(t1,t1) */ * from t1
join t2 on t1.a = t2.a

// the first will be chosen, and t2 is the build side
select /*+ NEST_LOOP(t2,t1) */ * from t1
join t2 on t1.a = t2.a

// the first will be chosen, and t2 is the build side
select /*+ NEST_LOOP(t2,t1), NEST_LOOP(t1,t2) */ * from t1
join t2 on t1.a = t2.a

// just like NEST_LOOP(t1, t2) + NEST_LOOP(t3)
// when join between t1 and t2, t1 will be chosen as the build side
// when join between the result after t1 joins t2 and t3,
// t3 will be chosen as the build side
select /*+ NEST_LOOP(t1,t2,t3) */ * from t1
join t2 on t1.a = t2.b
join t3 on t1.a = t3.c
```

Conflict between different hints

If multiple Join Hints are defined and conflict, they will be tried to apply one by one with the written order. If neither is satisfied, the planner's default join strategy is used. For example,

```
// throw an exception when parse this SQL
select /*+ BROADCAST(t1) */ /*+ SHUFFLE_HASH(t1) */ * from t1
join t3 on t1.a = t2.a

// the first hint BROADCAST will be chosen
select /*+ BROADCAST(t1) SHUFFLE_HASH(t1) */ * from t1
join t3 on t1.a = t2.a

// although BROADCAST is first, BROADCAST does not support full outer join,
// so choose the following join hint SHUFFLE_HASH
select /*+ BROADCAST(t1) SHUFFLE_HASH(t1) */ * from t1
full outer join t3 on t1.a = t2.a

// although there are two join hints defined,
// but all of them are neither support non-equivalent join,
// so choose the default strategy that planner infers
select /*+ BROADCAST(t1) SHUFFLE_HASH(t1) */ * from t1
```

```
join t3 on t1.a > t2.a
```

Dependent Processes

The use of Join Hint currently relies on the work that Calcite needs to be upgraded to 1.31. In Calcite 1.31, the following issue is fixed in it.

- Https://issues.apache.org/jira/browse/CALCITE-5107

In Calcite, more RelNodes are needed to implement hintable to realize the propagation of Join Hint.
Only when both the parent node and the child node implement Hintable, the hint will be propagated from the parent node to the child node.
For the specific implementation, please refer to the Calcite SQL Hint design document [4].

Compatibility, Deprecation and Migration Plan

Since this Flip only supports Join Hint to participate in the optimizer's choice of Join strategy, it has no effect on plan that did not have join hints before.

Test Plan

Each Join Hint strategy will be covered with UT tests and IT tests.

Rejected Alternatives

Enable propagating into subquery or view

In most popular databases and computing engines, hint only affects the current query block, and does not affect the join strategy in subquery and view. However, a syntax called QB_NAME is provided so that the join hint can be propagated to a view or subquery. We also conducted research based on this situation, please refer to Appendix [1] for more details.
However, this solution needs to introduce additional syntax of Query_Block_Name, which complicates the join hint syntax and behavior, so this proposal is not used in this flip temporarily.

Rejected resolving proposals when meeting conflict join hints

For join hint conflicts, we have many different behaviors to choose from. The following are the advantages and shortcomings of several behaviors. For specific examples corresponding to each behavior, please refer to Appendix [1].

	advantage	shortcoming

proposal 1: writing order first (we support)	Easy to resolve the hint conflict	The first hint by the written order may be not the most optimal strategy
proposal 2: high priority first (different join hints have different priority)	1. Always choose the better join strategy by the priority we define. 2. Users can write two or more two join hints and all of them can be considered by the actual scene.	1. Hard to define the priority when a new join strategy will be added. 2. Users can't control the order of trying different join strategies.
proposal 3: least cost first	1. always choose the faster join strategy by CBO 2. Users can write two or more join hints and all of them can be considered by CBO.	1. It depends on CBO and may lead to uncontrollable behavior. 2. Users can't control the order of trying different join strategies.
proposal 4: only try the first hint	Easy to resolve the hint conflict	The first hint may be not the most optimal strategy
proposal 5: ignore conflict hints	Easy enough to resolve the hint conflict	Sometimes user want to try both two join hints but this proposal will ignore all of them

References

[1]
https://docs.google.com/document/d/19PL77ZggPIhz7FMZv2Yx7w_cnYyvmTc-mxUpJYv_934/edit?usp=sharing
[2]
https://cwiki.apache.org/confluence/display/FLINK/FLIP-113%3A+Supports+Dynamic+Table+Options+for+Flink+SQL
[3]
https://cwiki.apache.org/confluence/display/FLINK/FLIP-204%3A+Introduce+Hash+Lookup+Join
[4]
https://docs.google.com/document/d/1mykz-w2t1Yw7CH6NjUWpWqCAf_6YNKxSc59gXafrNCs/edit#heading=h.y2wbp3wtyl2h