Aggregate Join Push-down

Zhong Yu, 2018-03-05

If an SQL query is an aggregate over a join, some aggregation work may be pushed down on branches of join, possibly reducing the execution cost. This relies on the property of the aggregate that it can be split over union and product. For example, for `count(*)`, we can push down counting on join branches, following the formula $\left|\bigcup_k (L_k \times R_k)\right| = \Sigma_k |L_k| \times |R_k|$

But that formula is incorrect in set theory... A note on duplicates. In this article we deal with sets of tuples, yet SQL tables allow duplicate rows. It's understood that to interpret a set as a table (base or derived), the table must have key columns, or one can be introduced. In the previous formula, we interpret L_k and R_k as base tables with keys, and the derived table $(L_k \times R_k)$ can be introduced with an extra key column with value k, therefore union $\cup_k$ will not de-duplicate any rows.

In this article, we first discuss aggregate push-down with general types of aggregate, grouping, and joining. We then apply the the general solution on SQL aggregate with group-by and inner equi-join, producing multiple equivalent queries that can be evaluated by a cost-based optimizer.

aggregate and group-by

An aggregate A applied on a table T, as AT, produces a table with a single tuple.

An aggregate A applied on a table T grouped by a a set of predicates G , as $A\overline{G}T$, produces a table defined by

$$A\overline{G}T := \bigcup_{g \in G} A\sigma_g T \tag{1}$$

As noted before, there won't be de-depulicate by $\bigcup$; the derived table has the same cardinality of G.

Multiple groupings is defined as

$$A\overline{G_1 G_2 \ldots G_n}T := \bigcup_{g_1, g_2, \cdots g_n} A\sigma_{g_1} \ldots \sigma_{g_n} T \tag{2}$$

aggregate push-down over union

Every aggregate $\bar{A}$ can be expressed as $\bar{A} = fA$ where A can be pushed down over union as

$$A \bigcup_k T_k = A^+ \bigcup_k AT_k \tag{3}$$

This can always be done; in the worst case, A simply retains all the rows, and A^+ restores them. Of course, in practice we are only interested in such A that does proper aggregation of data, e.g. `sum(T.x)`*.*

Since the result of any aggregate $\bar{A}$ can be computed from the result of an A with property (3), the rest of the article only deals with such type of A.

aggregate push-down over product

Every aggregate A over a product can be expressed as

$$A(L \times R) = a^*(A^L L \times A^R R) \tag{4}$$

Again, this is always possible; in the worst case, A^L and A^R can retain all the rows.

The nature of a^* is to map a tuple to another tuple. When a^* applies on a table with multiple rows, it's applied on each row individually, *i.e.*

$$a^* \bigcup_k T_k = \bigcup_k a^* T_k \tag{5}$$

L-R-composed predicates

We say that a predicate p is *L-R-composed*, if p is composed by p^L and p^R such that

$$\sigma_p(L \times R) = (\sigma_{p^L} L) \times (\sigma_{p^R} R) \tag{6}$$

theta-join partitioned

Every "theta-join" can be partitioned as

$$L \bowtie_\theta R = \bigcup_k \sigma_k(L \times R)$$

where every predicate k is L-R-composed (6), *i.e.*

$$L \bowtie_\theta R = \bigcup_k (\sigma_{k^L} L \times \sigma_{k^R} R) \tag{7}$$

In the worst case, the set of k simply enumerates every row in $\sigma_\theta(L \times R)$.

We want to partition (7) even further. Introduce an arbitrary set of predicates, P^L, such that any row in L satisifies one and only one predicate in P^L, therefore

$$L = \bigcup_{p^L \in P^L} \sigma_{p^L} L$$

Same for the R-side. Then,

$$L \bowtie_\theta R = \bigcup_{k, p^L, p^R} (\sigma_{k^L} \sigma_{p^L} L) \times (\ldots) \tag{8}$$

$(\ldots)$ *is the same formular as the L-side, but with "L" replaced by "R"*

aggregate over join with group-by

We now look at a query of the form

$$Q_1 := A\overline{G}(L \bowtie_\theta R)$$

From (1) and (8),

$$Q_1 = \bigcup_g A\sigma_g \bigcup_{k, p^L, p^R} (\sigma_{k^L} \sigma_{p^L} L) \times (\ldots)$$

We require that every predicate g in G is L-R-composed (6)

$$\sigma_g(L \times R) = (\sigma_{g^L} L) \times (\sigma_{g^R} R) \tag{R.1}$$

Then,

$$Q_1 = \bigcup_g A \bigcup_{k, p^L, p^R} (\sigma_{g^L} \sigma_{k^L} \sigma_{p^L} L) \times (\ldots)$$

Apply (3), (4), and (5)

$$Q_1 = \bigcup_g A^+ a^* \bigcup_{k, p^L, p^R} (A^L \sigma_{g^L} \sigma_{k^L} \sigma_{p^L} L) \times (\ldots)$$

query transformation

We now look at Q_2 of the form

$$Q_2 := A^+ \overline{G} a^* (A^L \overline{G^L K^L P^L} L) \bowtie_\theta (\ldots)$$

where G^L is the set of g^L. *(While a g^L may appear multiple times in G, G^L is a proper set without duplicates).* K^L is similarly defined.

We will prove that $Q_1 = Q_2$, under some requirements.

From (1)

$$Q_2 = \bigcup_g A^+ \sigma_g a^* (A^L \overline{G^L K^L P^L} L) \bowtie_\theta (\ldots)$$

We require that σ_g and a^* can be swapped, typically because a^* preserves columns used by g

$$\sigma_g a^* T = a^* \sigma_g T \tag{R.2}$$

From (R.1), (R.2), (7), and (2)

$$Q_2 = \bigcup_g A^+ a^* \bigcup_k \left(\sigma_{g^L} \sigma_{k^L} \bigcup_{g^{L\prime}, k^{L\prime}, p^L} A^L \sigma_{g^{L\prime}} \sigma_{k^{L\prime}} \sigma_{p^L} L \right) \times (\ldots)$$

We require that, typically because A^L, A^R preserves the columns used by g and k,

$$\sigma_{g^L} A \sigma_{g^{L\prime}} L = \sigma_{g^{L\prime} = g^L} (A \sigma_{g^L} L) \qquad \text{same for } k^L, g^R, k^R \tag{R.3}$$

Then,

$$Q_2 = \bigcup_g A^+ a^* \bigcup_{k, p^L, p^R} (A^L \sigma_{g^L} \sigma_{k^L} \sigma_{p^L} L) \times (\ldots)$$

Therefore, given (R.1), (R.2), (R.3), we have equivalency

$$Q_1 = Q_2$$

SQL aggregate, group-by, inner equi-join

We now apply the previous analysis to SQL query of the form

`select` A `from L, R where` $L.K_L = R.K_R$ `group by` $G_L \cup G_R$

and find it equivalent to

`select` $A^+ a^*$ `from L2, R2 where` $L.K_L = R.K_R$ `group by` $G_L \cup G_R$

with

`L2 := select` A^L `from L group by` $G_L \cup K_L \cup P_L$

`R2 := ...`

P_L, P_R are arbitrary column sets we can introduce.

Explanation: predicates g, k test that a row has specific values in G_L, K_L, G_R, K_R.

unique columns

If it is know that $G_L \cup K_L \cup P_L$ is a unique key of L, A^L always acts on a sigle row, therefore group-by is unnecessary in L2; instead we can do

`L2 := select` a^L `from L` a^L is non-aggregate, $\{a^L t\} = A^L\{t\}$

This can *always* be done regardless of G_L, K_L, because we can always choose P_L to contain the (possibly fictitious) unique rowId column.

This can be done on the R-side too of course. But if we do it on both sides at the same time, no aggregation is pushed down, the transformed query is close to, possibly identical to, the original form.

outer join

To handle left outer join, we may try adding rows of nulls on the R-side. However, this depends on whether A^R is sensitive to null rows. For example, `count(*)` counts all the rows, so this strategy won't work. But it may work for some type of aggregates. TBA.