SET OPERATIONS AND PROJECTION

To enjoy the full benefits of S™s, we need to adjust our architecture a bit. Let’s change the behavior of the query processor... calls to our API won’t be directly nested, but the returned S™s will be set to the best settings by the query processor before inserting to a next operator on input. Look at the following example:

```
select firstname, lastname from Users where firstname like 'Me';
```

Instead of calling:

```
project (select ("Users", { firstname, like, 'Me' }), { firstname, lastname })
```

The query processor can do:

```
S™ set = SetSmartestSettings (new S™ ("Users"));
S™ set = SetSmartestSettings (select (set, { firstname, like, 'Me' }));
S™ set = SetSmartestSettings (project (set, { firstname, lastname }));
```

In the SetSmartestSettings function, the query processor will set up the fetching and storing strategy to optimize the performance.

Lazy fetching should be directly supported in our API, as shown below. Of course, the query processor can decide not to lazy fetch, but to compute the results at once after the S™ is obtained. In the above example, the query processor would perform lazy fetching to apply both operators at the same time, effectively reading the relation only once.

The code below uses C# syntax. Functions supporting lazy fetching are annotated with [Lazy]. In this document, all discussed operators support it. That’s what the Context classes and Step functions are there for – the Step function performs a lazy step in the iteration and the Context class keeps the necessary information.

If an operator requires a sorted collection of tuples on input, the argument is annotated with [Sorted]. In such a case, the query processor is required to sort the tuples before calling the operator.

A) Operators

[Lazy]
```
[S™<T2>] Project (S™<T> set, ExpressionTree[] columns) {
    return new S™<T2> (ProjectStep, new ProjectContext<T> (set, columns));
}
```

[Lazy]
```
[S™<T>] Union ([Sorted] S™<T> set1, [Sorted] S™<T> set2, bool distinct) {
    return new S™<T> (UnionStep, new SetOperationContext<T> (set1, set2, distinct));
}
```
S^<T>\text{Intersection} ([\text{Sorted}] S^<T> set1, [\text{Sorted}] S^<T> set2, bool distinct) {
    return new S^<T> (\text{IntersectionStep}, \text{new SetOperationContext}<T> (set1, set2, distinct));
}

S^<T>\text{Difference} ([\text{Sorted}] S^<T> set1, [\text{Sorted}] S^<T> set2, bool distinct) {
    return new S^<T> (\text{DifferenceStep}, \text{new SetOperationContext}<T> (set1, set2, distinct));
}

B) Contexts

class ProjectContext<T> : Context<T> {
    S^<T> set;
    ExpressionTree[] columns;
}

class SetOperationContext<T> : Context<T> {
    S^<T> set1, set2;
    bool distinct;
    T last = null;
}

C) Iteration functions

T2\text{ProjectStep} (ProjectContext<T> context) {
    ... get next tuple – context. set. Next () ...
    ... allocate memory for a new T2 object ...
    ... project each of context. columns and copy it into the T2 object ...
    ... return the T2 object or null ...
}

T\text{UnionStep}<T> (SetOperationContext<T> context) {
    ... look at context. set1. Current (), context. set2. Current (), and context. last ...
    ... check the flag context. distinct and iterate to find the next smallest (non)distinct value ...
    ... return that value or null ...
}

T\text{IntersectionStep}<T> (SetOperationContext<T> context) {
    ... look at context. set1. Current (), context. set2. Current (), and context. last ...
    ... check the flag context. distinct and iterate to find the next smallest pair of identical values ...
    ... return that value or null ...
}

T\text{DifferenceStep}<T> (SetOperationContext<T> context) {
    ... look at context. set1. Current (), context. set2. Current (), and context. last ...
... check the flag context. distinct and iterate to find the next smallest value not present in set2 ...
... return that value or null ...
}

D) $S^m$s

Class $S^m<T>$

$S^m$ (string relationID) { ... for reading physical table ... }
$S^m$ (T[] tuples) { ... for reading in-memory array ... }
$S^m$<Tout, ContextType> (Func<Tout, ContextType> StepFunction, ContextType context) where
ContextType : Context<T> { ... for lazy fetching ... }

T Next () {
    ... either returns StepFunction (context) or asks for next row in a relation or next tuple in an array ...
    ... writes the result to a file, if ToBeStoredOnNext () is true ...
}
T Current () {
    ... last value...
}

bool IsLazy () { ... }
bool IsInMemory () { ... }
bool IsPhysicallyStored () { ... }
bool CanSeek () { ... }
bool CanSeekBackward () { ... }
bool IsSorted () { ... }
bool ToBeStoredOnNext () { ... }
void StoreAllNow () { ... }
string GetRelationId () { ... }
void SetPositionBefore (uint index) { ... }
T GetRow (uint index) { ... }
T[] GetRows () { ... }
uint[] GetRowIds () { ... }
bool IsColumnIndexed (uint index) { ... }

... add your favorite magic here ...