Item 110
git_comments:

1. * * This interface defines a non-ordered {@link org.apache.samza.sql.api.data.Relation}, which has a unique primary key * * <p> This is to define a table created by CREATE TABLE statement * * @param <K> The primary key for the {@code Table} class
2. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
3. * * Get the primary key field name for this table * * @return The name of the primary key field
4. * * Method to be invoked before the operator's output relation is sent * * @param rel The output relation * @param collector The {@link org.apache.samza.task.MessageCollector} in context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in context * @return The relation to be sent; return {@code null} if there is nothing to be sent
5. * * Defines the callback functions to allow customized functions to be invoked before process and before sending the result
6. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
7. * * Method to be invoked before the operator actually process the input tuple * * @param tuple The incoming tuple * @param collector The {@link org.apache.samza.task.MessageCollector} in context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in context * @return The tuple to be processed; return {@code null} if there is nothing to be processed
8. * * Method to be invoked before the operator actually process the input relation * * @param rel The input relation * @param collector The {@link org.apache.samza.task.MessageCollector} in context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in context * @return The relation to be processed; return {@code null} if there is nothing to be processed
9. * * Method to be invoked before the operator's output tuple is sent * * @param tuple The output tuple * @param collector The {@link org.apache.samza.task.MessageCollector} in context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in context * @return The tuple to be sent; return {@code null} if there is nothing to be sent
10. * * This method adds a {@link org.apache.samza.sql.api.operators.SimpleOperator} to the {@code OperatorRouter}. * * @param nextOp The {@link org.apache.samza.sql.api.operators.SimpleOperator} to be added * @throws Exception Throws exception if failed
11. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
12. * * This interface class defines interface methods to connect {@link org.apache.samza.sql.api.operators.SimpleOperator}s together into a composite operator. * * <p>The {@code OperatorRouter} allows the user to attach operators to a {@link org.apache.samza.sql.api.data.Table} or * a {@link org.apache.samza.sql.api.data.Stream} entity, if the corresponding table/stream is included as inputs to the operator. * Each operator then executes its own logic and determines which table/stream to emit the output to. Through the {@code OperatorRouter}, * the next operators attached to the corresponding output entities (i.e. table/streams) can then be invoked to continue the * stream process task.
13. * * This method gets the list of {@link org.apache.samza.sql.api.operators.SimpleOperator}s attached to an output entity (of any type) * * @param output The identifier of the output entity * @return The list of {@link org.apache.samza.sql.api.operators.SimpleOperator} taking {@code output} as input table/stream
14. * * Method to get the specification of this {@code SimpleOperator} * * @return The {@link org.apache.samza.sql.api.operators.OperatorSpec} object that defines the configuration/parameters of the operator
15. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-

2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
16. * * The interface for a {@code SimpleOperator} that implements a simple primitive relational logic operation
17. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
18. * * Method to be overriden by each specific implementation class of operator to perform relational logic operation on an input {@link org.apache.samza.sql.api.data.Relation} * * @param rel The input relation * @param collector The {@link org.apache.samza.task.sql.SimpleMessageCollector} in the context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context * @throws Exception
19. * * An abstract class that encapsulate the basic information and methods that all operator classes should implement. * It implements the interface {@link org.apache.samza.sql.api.operators.SimpleOperator} *
20. * * Method to be overriden by each specific implementation class of operator to handle timeout event * * @param timeNano The time in nanosecond when the timeout event occurred * @param collector The {@link org.apache.samza.task.sql.SimpleMessageCollector} in the context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context * @throws Exception Throws exception if failed to refresh the results
21. * * This method is made final s.t. the sequence of invocations between {@link org.apache.samza.sql.api.operators.OperatorCallback#beforeProcess(Relation, MessageCollector, TaskCoordinator)} * and real processing of the input relation is fixed.
22. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
23. * * This method is made final s.t. the sequence of invocations between {@link org.apache.samza.sql.api.operators.OperatorCallback#beforeProcess(Tuple, MessageCollector, TaskCoordinator)} * and real processing of the input tuple is fixed.
24. * * This method is made final s.t. we enforce the invocation of {@code SimpleOperatorImpl#getCollector(MessageCollector, TaskCoordinator)} before doing anything futher
25. * * The callback function
26. * * Ctor of {@code SimpleOperatorImpl} class * * @param spec The specification of this operator
27. * * The specification of this operator
28. * * Set of {@link org.apache.samza.sql.api.data.EntityName} as inputs to this {@code SimpleRouter}
29. * * Example implementation of {@link org.apache.samza.sql.api.operators.OperatorRouter} *
30. * * List of operators added to the {@link org.apache.samza.sql.api.operators.OperatorRouter}
31. get the operator spec
32. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
33. * * Set of entities that are not input entities to this {@code SimpleRouter}
34. * * Map of {@link org.apache.samza.sql.api.data.EntityName} to the list of operators associated with it
35. * * This class implements a simple stream-to-stream join
36. TODO Auto-generated method stub Do M-way joins if necessary, it should be ordered based on the orders of the input relations in inputs NOTE: inner joins may be optimized by re-order the input relations by joining inputs w/ less join sets first. We will consider it later.
37. TODO Auto-generated method stub
38. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
39. TODO: stub constructor to allow compilation pass. Need to construct real StreamStreamJoinSpec.

40. TODO Auto-generated method stub initialize the inputWindows map
41. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
42. * * This defines the base class for all keys used in window operators
43. * * The offset value in {@code long}
44. * * Helper method to get the maximum offset * * @return The maximum offset
45. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
46. * * An implementation of {@link org.apache.samza.system.sql.Offset}, w/ {@code long} value as the offset
47. * * Helper method to get the minimum offset * * @return The minimum offset
48. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
49. * * A generic interface extending {@link java.lang.Comparable} to be used as {@code Offset} in a stream
50. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
51. * * This class extends {@link org.apache.samza.task.sql.SimpleMessageCollector} that uses {@link org.apache.samza.sql.api.operators.OperatorRouter} *
52. * * Ctor that creates the {@code SimpleMessageCollector} from scratch * @param collector The {@link org.apache.samza.task.MessageCollector} in the context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context
53. * * This method swaps the {@code callback} with the new one * * <p> This method allows the {@link org.apache.samza.sql.api.operators.SimpleOperator} to be swapped when the collector * is passed down into the next operator's context. Hence, under the new operator's context, the correct {@link org.apache.samza.sql.api.operators.OperatorCallback#afterProcess(Relation, MessageCollector, TaskCoordinator)}, * and {@link org.apache.samza.sql.api.operators.OperatorCallback#afterProcess(Tuple, MessageCollector, TaskCoordinator)} can be invoked * * @param callback The new {@link org.apache.samza.sql.api.operators.OperatorCallback} to be set
54. * * Ctor that creates the {@code SimpleMessageCollector} from scratch * @param collector The {@link org.apache.samza.task.MessageCollector} in the context * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context * @param callback The {@link org.apache.samza.sql.api.operators.OperatorCallback} in the context
55. * * Method is declared to be final s.t. we enforce that the callback functions are called first
56. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
57. based on tuple's stream name, get the window op and run process()
58. ** * This example illustrate a use case for the full-state timed window operator *
59. 1. create a fixed length 10 sec window operator
60. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-

2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
61. filter all delete tuples before send
62. Now, connecting the operators via the OperatorRouter
63. 3. connect re-partition operator to the stream operator
64. * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, * software distributed under the License is distributed on an * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY * KIND, either express or implied. See the License for the * specific language governing permissions and limitations * under the License.
65. 4. create a re-partition operator
66. ** * This example illustrate a SQL join operation that joins two streams together using the folowing operations: * * a. the two streams are each processed by a window operator to convert to relations * b. a join operator is applied on the two relations to generate join results * c. an istream operator is applied on join output and convert the relation into a stream * d. a partition operator that re-partitions the output stream from istream and send the stream to system output * * * This example also uses an implementation of <code>SqlMessageCollector</code> (@see <code>OperatorMessageCollector</code>) * that uses <code>OperatorRouter</code> to automatically execute the whole paths that connects operators together.
67. 1. set two system input operators (i.e. two window operators)
68. filter all delete tuples before send
69. check whether the input is a stream
70. 2. connect join operator to both window operators
71. TODO Auto-generated constructor stub
72. * * Get the offset of the tuple in the stream. This should be used to uniquely identify a tuple in a stream. * * @return The offset of the tuple in the stream.
73. * * Get the message creation timestamp of the tuple. * * @return The tuple's creation timestamp in nano seconds.
74. * * Method to process on an input tuple. * * @param tuple The input tuple, which has the incoming message from a stream * @param collector The {@link org.apache.samza.task.MessageCollector} that accepts outputs from the operator * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context * @throws Exception Throws exception if failed
75. * * Method to initialize the operator * * @param config The configuration object * @param context The task context * @throws Exception Throws Exception if failed to initialize the operator
76. * * Method to perform a relational logic on the input relation * * <p> The actual implementation of relational logic is performed by the implementation of this method. * * @param deltaRelation The changed rows in the input relation, including the inserts/deletes/updates * @param collector The {@link org.apache.samza.task.MessageCollector} that accepts outputs from the operator * @param coordinator The {@link org.apache.samza.task.TaskCoordinator} in the context * @throws Exception Throws exception if failed
77. TODO Auto-generated constructor stub
78. * * A simplified constructor that allow users to randomly create <code>PartitionOp</code> * * @param id The identifier of this operator * @param input The input stream name of this operator * @param system The output system name of this operator * @param output The output stream name of this operator * @param parKey The partition key used for the output stream * @param parNum The number of partitions used for the output stream * @param callback The callback functions for operator
79. * * A simplified version of ctor that allows users to randomly created a window operator w/o spec object * * @param wndId The identifier of this window operator * @param lengthSec The window size in seconds * @param input The input stream name * @param output The output relation name
80. 3. connect re-partition operator to the stream operator

git_commits:

1. **summary:** SAMZA-552 update operator APIs
   message: SAMZA-552 update operator APIs

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Implement window operator in Samza
   description: The discussion is based on how to support tuple and/or time based window operators in Samza physical operator layer. Here are the few observations: # Tuple represents the “physical ordering” of events while time-based window has semantic meanings to users # Total ordering between tuples are possible within Samza/Kafka given a deterministic MessageSelector on all input streams and offsets within each stream # No matter whether tuple or time is used to measure the window size, the window termination condition is needed to close a window to avoid the job to be wedged forever The following questions have to be answered to fully implement a window operator: # how to determine that a window is closed and no new tuples will be added? ## For tuple based, how do we close the window if messages do not come or get delayed? ## For time based, how do we close the window if ### the messages are not strictly in order w/ the time? ### the message w/ timestamp greater than the window boundary does not come or gets delayed?

jira_issues_comments:

1. Some reference: http://cs.brown.edu/~ugur/streamsql.pdf And one more question: how do we support the specification of the window termination condition? In SQL grammar extension or some system config?
2. This talk is fantastic: http://www.infoq.com/presentations/millwheel
3. **body:** Yeah one thing it reminded me of was an idea I've had a couple times with respect to windowing, basically just treating everything as updatable as a way to handle late data. Once you get rid of the idea that grouping over a window produces only a single output for the group key the whole problem of late data is kind of solved. The actual time scheme in MillWheel has always struck me as a bit complex. But this talk reminded me of the same idea, which is anything on event time windows needs to be updateable. I'm not sure you even need their whole marker/trigger/watermark scheme. Consider a case where you are doing select trunc_to_minute(time) min, user_id, count(1) from page_views group by trunc_to_minute(time), user_id; Imagine that you are using our key-value store to capture count data as it occurs by doing put(user_id + minute, get(user_id + minute) + 1) Say you have input Time, User 12:01:01, 1 12:01:03, 1 12:01:05, 2 12:01:09, 1 In the absence of any caching the changelog stream would basically capture something like 12:01, 1, 1 12:01, 1, 2 12:01, 2, 1 12:01. 1, 3 In other words, the stream of updates. The whole windowing scheme they propose is just a way of suppressing some of those updates so that you give just one count if there is no late data. But effectively you can view the caching layer in the key value store as doing a similar thing--attempting to supress duplicate updates until the commit point. If the downstream thing needs to know to treat the data like a table that is updated, which it will have to do if you ever admit the possibility of late data, then it will have to handle updates. So in that case the pruning of duplicate updates is really just a performance optimization. The configuration you have to provide is what TTL to have in RocksDB because at some point you are just going to toss the old counts and at that point you really can't take anything later, but that could be days.
   label: code-design
4. One of the nice things about this is that it pushes the handling of late data downstream. The consumer of the results can choose how long they want to suppress display of the aggregate to consider it "complete". They could choose to show the aggregates immediately in which case you would kind of get to watch the counter increase as more data arrived, or you could suppress until the counter is supposed to be complete and then use it. Likewise you can choose whether to accept or reject one of these updates--that is the user of the data gets to choose when the "close of books" occurs, which seems better to me.
5. **body:** One thing that I found slightly odd about the description of late data in the talk was that it seems that their solution only works for some use cases. For example, if you are doing a LEFT JOIN between two streams "a" and "b". At some point, you will emit the results. If a message has arrived for "a", but not for "b", then you'll emit "a", NULL. If (within the 2 week time span described in the talk) the corresponding "b" arrives, you'd then emit "a","b". Now your stream has: "a", NULL "a", "b" Now suppose you had two counters: "hits" and "misses". You increment hit every time there's no null, and miss every time there's null. You'd end up with hits=1, misses=1. You'd expect to get hits=1, misses=0, though. One way around this specific example would be to know that "a","b" is due to a late arrival. This would let you decrement the misses column. A more sophisticated example would be hits, misses_a, and misses_b. In such a case, not only would you have to know that "a", "b" was due to a late arrival, but you'd have to actually know the last message as well ("a", NULL) -> ("a", "b"). I wonder if this is something Millwheel provides that just wasn't talked about. I wonder if you construct even more complex use cases where this model fully doesn't provide exactly-once semantics. I like that they simply said, "strong consistency," not, "exactly once." That gives you a bit of wiggle room in these scenarios. bq. Yeah one thing it reminded me of was an idea I've had a couple times with respect to windowing, basically just treating everything as updatable as a way to handle late data. Ya, we'd been discussing this internally. It's a little mind-bending for end users, I think. Doesn't mean we can't do it. Just means we'd have to provide a lot of good clear description about it. bq. The whole windowing scheme they propose is just a way of suppressing some of those updates so that you give just one count if there is no late data. Agree. Another interesting thought is if we could integrate in some way with audit, to allow us to know "for sure"-ish when a window is closed. bq. Likewise you can choose whether to accept or reject one of these updates--that is the user of the data gets to choose when the "close of books" occurs, which seems better to me. Yea. The way we'd been talking about windowing, I think this will work. We'd been thinking of late arrivals (after punctuation) resulting in a drop. But if the state is still persisted, there's nothing that says we couldn't emit the "updated" late arrival value. Should be flexible. One of the main advantages that Millwheel has is that it can store 2 weeks worth of state. It's using a remote store (BigTable). Since we're local, this

might be a bit large. For aggregation, it's probably do-able, but for large join jobs, 2 weeks worth of state might be many TBs of space. 100k msgs/sec@200 bytes/msg * 14 days = ~24 TB. Not un-doable, but would require a fair number of machines to do this. If the 2 week state has a changelog attached, then it's both local and in Kafka (with replication).
label: code-design

6. Observation: I also think that what they describe is essentially just a 2 week sliding window. [~julianhyde] had described sliding windows as having the property of preserving cardinality between input and output. If you take their example remove the partial-suppression that they're doing (1 minute stream time, or watermark move time), you just have a 2 week sliding window. How you suppress determines the characteristics of the sliding window. You could turn it into a tumbling window, a hopping window, etc.
7. **body:** [~criccomini] I think your example is just another way of saying that the output stream is fundamentally a stream of updates with some "best effort" suppression of duplicate output. Your stream does have both "a", NULL and "a", "b" but the interpretation of that stream is "first I thought ('a',NULL) but then I revised that to ('a','b') once I got more information". So it's not a duplicate, it's an update. I agree that this view of windowed aggregates might be confusing but I once I got it I actually didn't mind it. In some sense emitting the running count is a very natural fit for streaming and similar to what you as a human would do. In terms of efficiency I am less concerned. Two weeks is insanely late, I think 4 hours is a little more reasonable. After all with the "punctuation" model you would be doing 5 minute aggregates and only giving like a few minutes before you output the result and close the books on that window. That time period is equivalent to the retention here, since you will have to drop any event outside that window. Basically the implementation would be that you have a configurable setting event.rejection.horizon=2 hours and anything after that you drop. So for people who want efficiency they would set that to just a few minutes but for people who want accuracy at all costs you would set it to a day.
label: code-design
8. **body:** Jay, when you're talking about a rejection horizon, is that wall-clock-time? I'm curious how you see this method working when replaying old data. Let's say I have a month of raw input data in kafka and whenever I create a new model I run it over that month and compare output to some source of ground truth. It seems like this would cause two problems: 1. The local state becomes unreasonably large as you keep around weeks worth of 5 minute windows, since you're going through that month much faster than 'real-time'. 2. The processing of some partitions could proceed significantly faster than other partitions, so if there's some sort of join/merge you can pile up state waiting for a partition doing heavier processing to catch up.
label: code-design
9. Attaching a draft on window operator that we have been discussing internally in LinkedIn. I wrote this before reading through the MillWheel talk and here are a few comparisons to MillWheel strategy: 1. MillWheel "stream time" is documented as "system time" in this draft 2. The punctuation mark in the draft is equivalent to the "trigger" in MillWheel talk 3. The window size measurement in the draft is a specific example of "watermark" in MillWheel. The followings is the interesting concept from MillWheel that we are planning to extend our model to support: 1. Re-emit window results for late arrivals. This would include two parts: a. extend the current window state to a store that keeps all past windows; b. add policies to allow late arrivals to trigger re-emitting results for a past window. With the above extension, we will be able to fully incorporate MillWheel's model in the draft design.
10. The reprocessing case is an interesting example. It seems that this is essentially the same problem that Kafka wrestles with for its time-based/size-based retention policies. It seems like we'll want something similar here as well (purge windows while oldest window > 4h old, or total size > 500GB).
11. Thinking on this more, you might want to use event time, not stream time, to define the retention. That way, if you reprocess, and get 2 weeks of data in 5 minutes, you'd just be aggressively trimming your state based on event time within the 2 week stream.
12. **body:** Controlling retention with event time works fine, I think, as long as you don't have interleaving of streams? A step where one stream provides messages to many partitions in another topic can remove the independence of steps. If your state is relatively small, this isn't such a big deal, you can just have an extra long retention period (long enough to cover the skew and late data). But if your state is large, this seems like it would get problematic. Diagram: http://i.imgur.com/DTmfvad.jpg?1 I can provide a more concrete example of my concerns if it would help.
label: code-design
13. LOL, I love that diagram. :) If you have interleaving of streams, then you don't have an exact time. You have to have some heuristic for determining it, such as the max(event time) you've seen, or the average of the highest N, or something. I agree, if your skew (not necessarily state) is large, this could be quite problematic.
14. Just to document some of the discussion items on the MillWheel model: We did not think about handling late arrivals as a "correction" to the earlier results. If we follow that model, the late arrival problem can be addressed in the following: a) Each window will have a closing policy: it would either be wall-clock based timeout, or the arrival of messages indicating that we have received all messages in the corresponding event time window b) Each window also keeps all the past messages it receives in the past windows, up to a large retention size that covers all possible late arrivals c) When a window's closing policy is satisfied, the window operator always emits the current window results d) When a late arrival message came, the window operator will re-emit the past window results to correct the previous window results Looking at a windowed aggregation example: {code} SELECT FLOOR(event_time TO HOUR) as time, COUNT(*) as counter FROM Stream GROUP BY FLOOR(event_time TO HOUR) {code} In this example, the aggregation for the counter for window from 10:00-10:59 will have a "wrong" value when the window is closed by an arrival of message w/ 11:00 timestamp, but will be corrected later by a late arrival of another message in the time window from 10:00-10:59. I.e. if we keep all the previous window states, late arrival messages will simply trigger a re-

computation of the aggregated counter for the window 10:00-10:59 and overwrite the previous result. In this model, the final result is always correct, as long as the late arrivals is within the large retention size. It seems that the followings are more reasonable: 1) Window operator will have a full buffered state of the stream similar to a time-varying materialized view over the retention size 2) Window size and termination (i.e. sliding/tumbling/hopping windows) will now determine when we emit window results (i.e. new messages/updates to the current window) to the downstream operator s.t. the operators can calculate result in time 3) Late arrivals will be sent to the downstream operator and triggers a re-computation of the past result based on the full buffered state 4) Optionally, the downstream operator can decide to suppress the output for a while to allow late arrivals to correct the result before sending the output. In the above model, the window operator becomes a system feature, or an implementation of "StreamScan" in Calcite's term. And we do not need specific language support for the window semantics, with a default time window operator implementation that serves as a "StreamScan". All window definition in the query language now only dictates the semantic meaning of aggregation and join on top of the physical window operator which provides: a) a varying/growing materialized view; b) a driver that tells the aggregation/join to compute/re-compute results on-top-of the materialized view. I will update the design doc to reflect the above thoughts.

15. Hey Yi, two quick questions where I'm not sure if I follow: 1. Why is there a concept of "window close" at all? Once you allow for corrections you are basically treating a window as mutable state and issuing corrections. So why not just view the entire computation of the window as a running stream of updates from beginning to end. There is no real difference between updates before the window close and after the window close. Can't we just get rid of the whole concept of window close? Whenever an update comes in it updates something and that logs out the new value, this is true whether you are currently in the window or whether (according to some clock) the window has "closed". All the complexity of the millwheel model can just go away. The only real "close" concept that exists is the retention window...once you get updates to data outside your retention window you have to toss it out. I agree that the problem of what clock to base the retention window on remains. Basically if I understand what you are saying I think you are still trying for a model where we attempt to issue a single canonical output for a window, and then correct it only if late data arrives. But what I am advocating is just using the existing state storage mechanism for the aggregate and having the changelog for that be the result of the query. Since you have thought this through more I think the question I was hoping to see answered is whether we can just make the hard assumption that "aggregates" == "key value store changelog" with no additional support needed? I think this matters because if you try to avoid output until the end of the window, you will need to have some backup for the state in the key-value store, so you will still need a changelog. But that means having both a changelog and an output stream with almost identical contents, right? 2. In general the materialized result in the k/v store will be the aggregate not the raw rows, right? I.e. in your sql example we would be storing the mapping hour=>count not hour=>{events}. I think this is what you are saying, but I'm not sure. So in the counting case there is no real "recomputation" it is just a matter of processing the new row and outputing the final value (count+1). I think this is often true, but maybe not always? Maybe joins and non-incremental aggregates like median are counter-examples? Perhaps this is just a sort of optimization done in the query planning layer to push down the aggregation all the way to make the aggregate itself incremental even though the value is in some sense equivalent to storing all the rows and recomputing from scratch on each update?
16. **body:** I think 1 is the simplest answer, it's very elegant if your state fits that model. My personal counter-example is a join in physical space where multiple points can (as a weighted average) contribute to a final output point. In this case it's highly desirable to avoid smoothing even twice, so smoothing every time a new bit of contributing data arrives would be very undesirable.
    label: code-design
17. **body:** Yeah I think you are right, this is true I guess for any non-incremental computation. Median would be the same way--a simplistic implementation would require a full scan of the rows for the exact computation on each update. However those cases are somewhat rare (?) and the cost of having an entirely seperate log plus all the complexity of punctuation etc definitely adds both mental and computational burden in the common case where computation is incremental.
    label: code-design
18. **body:** Hi, [~jkreps], let me comment below: # You are right about "window close". What I really meant for is that an event or a timer to trigger *window update* for a particular window. Some types of windows can truly be closed, such as a fixed length tuple window. But that would simply means no more updates for that window to the downstream operators. Regarding to the "single canonical output for a window", I am not very strong about that either. The window update should be triggered either a) some system time governed deadline demands a timely output; Or b) there is "enough" data in the window to trigger a meaningful calculation. The whole purpose of "waiting till the end" is trying to avoid too many window updates that could be consolidated otherwise. # As for the content stored in the state store, here are the thoughts that I have now: a) "the aggregates == key value store changelog" probably means that the key value store has a unique key per window aggregation, and we just update the store for every update comes in? I agree that would be much more efficiency in state management. However, as you already commented, it does not work for the following cases: ## aggregations that need all samples in the window, such as median, 99-percentile ## joins that need to access each messages in a past window Hence, I would rather focus on designing a general state store for the window operator first. The aggregations that does not need full sample set in the window can be considered later as an optimization. # I agree with you that some aggregation can be pushed down to the window operator as an optimization. But for the above reasons, I don't think that we can make a general case for that. Thanks!
    label: code-design
19. Hi, [~geoffry], thanks for the example. This issue can be address by controlling when the window operator sends the update via a policy. For your example, it may be desirable to configure the window operator wait till a) a timer expires;

Or b) we have accumulated enough messages in a window that we conclude that the window "ends".

20. **body:** I think we are in agreement about pushdown. Incremental operations can store the value so far and in that case the changelog is the output. This will cover the common case of sums, counts, max, min, and other projections. What is the right mathematical term for this property? I think a full outer join has this property but other joins don't. Medians and quantiles don't have the property as you say though they are not too important. However in those common cases the aggregation would just be a normal state store containing the result so far, right? In the non-optimizable cases we are still talking about using the existing key-value store, right? It is just that the content of this store isn't the output of the operator. This would be the case for an inner join, I think, where the contents of the store are the rows in the window but the output is the join computed off this state. I think it would be good to use (or improve) the existing abstraction so that we can put some time into really nailing performance on that and have it pay off for both direct Java usage of the key value store and also the operator usage.
    label: code-design
21. Yes, I agree that it just seems as that the content of the window would differ at a high-level view. The issue that makes me deviate from this high-level abstraction is the case of sliding windows. If we follow the abstraction of storing the following: (windowKey, windowOutput) where windowOutput can be optimized to an aggregated result. For sliding windows in a inner-join, many entries in two adjacent sliding windows are the same and there would be huge amount of redundant data. And in my mind, sliding windows for inner-join seems to be a common use case. In this case, it is more efficient to use a single store that keyed by eventtime / offset and allow access to the past windowOutput as a range query to the store. The actual difference in the underlying store may be hidden by an interface of getWindowOutput(windowKey).
22. Yeah I agree that the contents and usage of the store will vary but won't we still be using the same org.apache.samza.storage.kv.KeyValueStore abstractions to implement these different operations? If not that is unfortunate as it will make any optimization difficult.
23. Ah, I see. I misunderstood your concern. Of course I am still thinking of using the same KV-store abstraction. Totally agreed on this point.
24. Adding updated design doc. Highlights: 1. Fully adopt MillWheel model to allow multiple updates to the window outputs 2. Separated the design between the window usage in aggregation and join 3. Added more data structure/data store design details
25. Fixed the reference links.
26. [~jkreps] and [~milinda], if you can review and comment on the latest design doc, it will be greatly appreciated. Thanks!
27. **body:** The overall design/concept looks good to me. I'm mostly interested in the API, which is listed at the very end. Is the API at the end of the doc meant to be used by users, or by an query layer? I found that requiring both time and offsets in this API are a little confusing.
    label: documentation
28. Sorry for the late reply. Design looks good to me. I'll integrate this to query planning design document. P. S. Given that we have confluence wiki for Samza, I think we should move all Streaming SQL related design docs to that. WDYT?
29. [~criccomini], thanks for the comment. As for the API, I separated the APIs into two sets: one for automatic query layer, and some additional APIs for standalone operators used by the user directly. {quote} I found that requiring both time and offsets in this API are a little confusing. {quote} I assume that you were referring to the getMessages() API here. It makes sense to not mix time and offsets. We can actually only use time in time-based window and only use offset in tuple-based window. I will make the update to the APIs here.
30. **body:** [~milinda], thanks! [~criccomini], what's our way to keep all our design docs? Confluent wiki seems to be a new thing that is not documented here: http://samza.apache.org/contribute/design-documents.html
    label: documentation
31. IMO, the most important thing is to not scatter the design docs all over the place. Currently, all design docs are on JIRAs. You can raise this for discussion on the mailing list.
32. [~criccomini], if the JIRA is the single place for all design docs, we should follow. I just updated the JIRA w/ design labels s.t. it shows up in the list [here|https://issues.apache.org/jira/issues/?jql=project%20%3D%20SAMZA%20AND%20labels%20%3D%20design%20ORDER%20BY%20priority%20DESC]. Confluent links can be added to the JIRA s.t. we don't lose track of those docs.
33. Updated based on [~criccomini]'s comments.
34. RB: https://reviews.apache.org/r/34206/. This patch fixed some Java doc issues.
35. Merged to samza-sql based on RB review.
36. RB: https://reviews.apache.org/r/34500/ Somehow the RB board email notification failed to work. Hence, re-posting the RB link here to solicit review feedbacks. Thanks!