Distributed Order Dependency

Sebastian Schmidl, Juliane Waack[1]

Abstract: Abstract goes here.

Keywords: Actor Model; Akka; Distributed Computing; Parallelization

1 Introduction

1.1 Order Dependencies

- What are ODs?
- Why are they useful?
  - Query optimization
  - Data quality (find constraints, help understand semantics)
  - support index selection

1.2 Order Dependency Discovery

What are the challenges?

- Defining Minimality (Unhelpful definition of minimality in Naumann paper, incorrect one in Consonni paper)
- Large searchspace (Long runtime and large memory-needs)
- current agorithms slow on large datasetz
- Therefore need scalability -> distribution

1.3 Distribution

- What are distributed Systems? Why are they useful?

[1] Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, {sebastian.schmidl, juliane.waack}@student.hpi.de

- What is the actor model? Why use it here?

2 Related Work

The concept of order dependencies was first introduced in the context of database systems by Ginsburg; Hull [GH83] as *point-wise ordering*. Ginsburg; Hull's definition specified that a set of columns orders another set of columns.

Szlichta et al. [SGG12] later introduced another definition for order dependencies, which was used by the following research in this area [Co19; LN16; Sz17] and is also the basis of this work. It differentiates from the point-wise ordering by considering list of columns instead of sets. This leads to a lexicographical ordering of tuples, as by the `order by` operator in SQL.

Langer; Naumann published the first algorithm to find all minimal order dependencies based on Szlichta et al.'s definition. It is called `order` and has a factorial runtime.
When publishing their own algorithm `fastod`, Szlichta et al. showed, that Langer; Naumann's definition of minimality was incomplete. `fastod` has an exponential worst time complexity [Sz17].
When introducing the algorithm `ocddiscover`, Consonni et al. claimed, that a provided implementation of `fastod` did not find all order dependencies. `ocddiscover` has a factorial runtime but was shown to outperform `fastod` on some datasets. It is also capable of multithreating, which makes it easy to adapt to a distributed system. This was the state of published research when we started on this project, which is why we based our own algorithm on `ocddiscover`.
Since then, Szlichta et al have published an Errata Note demonstrating an error in Consonni et al.'s proof of minimality and explaining, how Consonni et al. misinterpreted `fastod`'s results. Since the goal of our seminar was the distribution of an already existing algorithm with a focus on reliability and scalability, however, we decided to continue working on `ocddiscover`.

3 Approach

3.1 OCDDISCOVER

The OCDDISCOVER algorithm creates the searchspace for finding *order dependencies* over *order compatibilities*. This can be done, because they prove that whenever there is an *order dependency*, there is also an *order compatibility* and a *functional dependency*. Using *order compatibilities* reduces the search space, because they are symmetrical.
Consonni et al build the search tree, by testing each candidate for being order compatible. If it is not, no children are generated for this candidate. If it is order compatible the candidate

is checked for being order dependent. If it is we have found a minimal order dependency and no children need to be checked, as they would not be minimal. If it is not, children to be checked later are generated. This happens by adding a column that does not yet appear on either side of the candidate once to its left and once to its right side.
Consonni et al proved in their paper, that they would be able to find all minimal *Order dependencies* using this method of generating candidates. However, as Szlichta et al showed in their Errata Note, Consonni et al made a mistake in one of their proofs and this method of building the searchspace does not create minimal *Order dependencies* with repeated prefixes.

3.2 Our architecture

3.2.1 System architecture

The cluster we use has a *Peer-to-peer* architecture. Every node has the same structure and works on the same kinds of tasks. Every node also hold the complete dataset, because the checking of *Order Dependency candidates* requires access to the contents of all columns. The *Peer-to-peer* architecture allows every single node to fail without work getting lost or the algorithm needing to be restarted.
The only circumstance during which this is not the case is when the systems is newly started. At the beginning a single node, which we call *Seed node*, is responsible for loading the data, the initial pruning step and the generation of the first set of candidates. If the *Seed node* were to fail during these first steps, the cluster would need to be restarted. As they usually only take a few seconds, though, this seemed to be a reasonable constraint.
To begin working, every node needs the dataset, the information which columns can be pruned and a first set of candidates to check. All nodes that are not the *Seed node* can get the dataset from another node via streaming. Once it is done with pruning, the *Seed node* sends the pruning results to all other nodes in a broadcast. A node that joins later can ask any other for this information. The distribution of candidates from the *Seed node* to all other nodes in the cluster is done using *Work Stealing* (3.3.1), which is described in more detail later.

3.2.2 Node Architecture

Every node in the cluster shares the same actors. These are both helpful for parallelizing the tasks each node work on as well as structuring the communication in between nodes.

Master The *Master actor* is responsible for holding the state of a node. In contains the *Candidate queue* and *Pending queue* and distributes the candidates to the *Worker actors* to be worked on in parallel. We use a pull-based apporach for this distribution of tasks. This means that the *Master* only ever responds to *Workers* asking for candidates and therefore does not have to monitor the state of every single *Worker* of the node. The *Master* also deals with getting new candidates using *Work Stealing* (3.3.1), when the *Candidate Queue* is empty and initiates the node shutdown when the *Downing Protocol* (3.3.2) reveals that all *Order Dependencies* have been found.

Worker A node can have an arbitrary number of *Worker actos*, though the number of available threads - 1 is usually a good estimation for effective ressource usage. The *Workers* check candidates, which they get from the *Master* and generate new ones, which they send back for the *Master* to append to the *Candidate Queue*. This checking and generating of candidates is the main task of each node and therefore requires the most ressources. All *Order Dependencies* and *Order Compatibilities* the *Worker* finds are send to the *Result Collector* to be recorded into a file.

Data Holder The *Data Holder actor* is responsible for loading and holding the dataset, for which the *Order Dependencies* are to be found. A reference to this dataset is shared with the *Workers* who need it to check candidates. If a path to the dataset is provided to the node on start up, it loads it from the file. If not, the *Data Holder* asks one of their neighbours in the cluster to provide them the dataset via streaming. Once the dataset is loaded, the *Data Holder* notifies the *Master* that the node is ready to start working on checking candidates.

Result Collector The *Result Collector actor* gets send all found *Order Dependencies* and *Order Compatabilities* by the *Workers*. It collects and counts these results and writes them to a file so they can be recovered even if the node fails unexpectedly.

Cluster Listener The *Cluster Listener actor*'s task is to determine and share who the neighbors of its node are. Whenever a new node joins the cluster all previous send it references to their *Master* and *State Replicator actors*. The new *Cluster Listener* sends back the corresponding information for its node.
When the ordering of nodes changes because a node joined or left the cluster, the updated neighbours are send to the *State Replicator*. In contrast, the *Data Holder* only receives information about the neighboring nodes, when it specifically asks for them, because it only needs the information in the beginning to know whom to ask for the dataset.

State Replicator The *State Replicator* holds the state of its neighboring nodes and shares the *Master's* state with them. This enables us to recover work from unexpectedly failed nodes and is explained in more detail in the *State Replicaton Protocol* (3.3.3)

Reaper The *Reaper Actor* watches all other actors and cleanly shuts down the system once all of them are terminated.

3.3 Communication Protocols

3.3.1 Work Stealing Protocol

When a node is out of work, either because it just joined the cluster or because all the candidates it checked got pruned and did not generate any new candidates to check, it tries to take over some of the work of the other nodes. This process is called *Work Stealing*. We use it to ensure that no node is idle while the others are still checking candidates and to balance the workload over the cluster. The workload of a node is defined by the number of candidates waiting to be processed in its queue. In the desired outcome to add to its own queue of the *Work Stealing Protocol*, each node ends up with a similar workload. To achieve this goal the *Master Actors* of the different nodes communicate their workloads with each other.
The *Work Stealing Protocol* is initiated when a node's *Candidate Queue* is empty, even if some of its workers are still processing candidates and might return new ones soon. This node (the *Thief node*) sends a message to all other nodes, asking for the current size of their *Candidate Queues*. It also sets a timeout after which it processes the answers of the other nodes. During this processing step it calculates the average *Candidate Queue's* length of all the nodes that answered and itself. To improve the balancing of workloads over the cluster, the *Thief node* only takes candidates from nodes that have a *Candidate Queue* of above average length and only takes so many candidates that its own *Candidate Queue's* length does not exceed the average. To get the candidates from another node's *Candidate Queue*, the *Thief node* sends a message with the amount of candidates it wants to take to the node it wants to take them from. This node, the *Victim node*, then moves the asked for candidates from its *Candidate Queue* into its *Pending Queue* and sends them to the *Thief node*. However, since the *Victim node's* queue might have changed since it first sent its length to the *Thief node*, the *Victim node* sends back at most half of the candidates in its queue. Once the *Thief node* receives the candidates, it adds them to its own queue and sends a message acknowledging the receipt to the *Victim node*, which then deletes the candidates from its *Pending queue*. To ensure the send candidates arrive and will get processed, the *Victim node* watches the *Thief node*. In the case that the *Thief node* failes before it could acknowledge the receipt of the candidates, the *Victim node* moves them back to its own *Candidate queue* from its *Pending queue*.

3.3.2 Downing Protocol

When all *Order Dependencies* have been found and there are no more candidates to check, the system should shut itself down. A node starts the *Downing Protocol*, when it has no more candidates in its *Candidate Queue* and *Pending Queue* and an attempt to get more work using *Work Stealing* was unsuccessful. Checking the *Pending Queue* is important because candidates currently being processed could generate new candidates and therefore new work.

To ensure that there are no more candidates to be processed and no more candidates being generated in the whole cluster, the node has to ask every other node, whether they are also out of candidates. If another node still holds unprocessed candidates or is still in the process of creating new candidates, these could be redistributed over the cluster using *Work Stealing* so the node can continue to work instead of shutting itselves down. The node waits for a response from every other node in the cluster, also being aware of nodes leaving the cluster and subsequently not awaiting their response anymore.

If any of the other nodes respond with a message saying that they are still holding or processing candidates, the node switches from the *Downing* to the *Work Stealing Protocol*. If all nodes have either left the cluster or responded that they have no more work, the node shuts itself down.

3.3.3 State Replication Protocol

To ensure any of the nodes of the cluster can fail without any work getting lost, every node regularly replicates its state to two other nodes. Because of this, three specific nodes would have to fail simultaneously to permanently lose any unprocessed candidates. In this case, everything would need to be restarted to ensure complete results.

Which nodes the state is sent to is decided by their addresses. All nodes in a cluster are ordered by their addresses and every node replicates its state to both its neighbours in this ordering. The first and last node in this ordering are also each others neighbours.

Every node has a *State Replication Actor* responsible for regularly sending the node's state to its neighbours and processing these neighbour's states when they are sent. It gets the information of who its neighbours are from the *Cluster Listener Actor*.

In regular intervals, the *State Replicator* asks the *Master Actor* for its current state. The state is a combination of its *Candidate Queue* and *Pending Queue*. On receiving the *Master's* current state, the *State Replicator* sends to both its neighbours with a version number that is continually increased.

When a *State Replicator* receives the state of one of its neighbours, it checks whether the version number in the new message is higher than the version number of the last state it received from that actor. If this is the case, it saves the new state and version number. If not, is discards the new message, because it already has a more up to date state for that actor.

A node joins the cluster When a new node joins the cluster, the *Cluster Listener* updates its ordering of all the nodes in the cluster. It then sends the *State Replicator* the updated neighbour information. If the nodes neighbour has changed, the *State Replicator* removes the state of the node that is no longer its neighbour and sends its own state to its new neighbour.

A node leaves the cluster When a node leaves the cluster, its two neighbours have to decide which of them will take on the work the leaving node did not finish. They are alerted to their neighbour's leaving by the *Cluster Listener* who also immediately tells them, who their new neighbour is. This new neighbour was the leaving nodes other neighbour. To determine which of the two has the most up to date state of the leaving node, they share the version number of the leaving node's state. The node with the higher version number adds the leaving nodes state to its own *Candidate Queue* to ensure all candidates get processed. The node with the lower version number deletes the state of the leaving node. The two nodes then share their state with each other.

4 Evaluation

- What to test? (see testing-strategy)
- Testing setup
- Tests and results

5 Conclusion

- Summary
- real conclusion
- future work
  - what's not implemented yet (merging of results, deduplication, fixing algorithm)
  - optimize (sort and compare on multiple columns (consider data types or merge values to one or define ordering over indices?))
  - interesting questions: how to proove we have founbd all edgecases / formalize protocols
  - Test memory boundaries, possibly optmize memory consumption (delta protocol?), look at garbage collector overhead
  - Find and fix work stealing edge cases
  - evaluate timeouts
  - The actor model provides a dynamic aspect. This can be used to run multiple approaches in a single system and to exchange information between them. The

goal of this would be to benefit from the advantages of the different approaches and at the same time decreasing their disadvantages. Our approach does not make use of this aspect of the actor system so far.

References

[Co19] Consonni, C.; Sottovia, P.; Montresor, A.; Velegrakis, Y.: Discoverying order dependencies through order compatibility. In: International Conference on Extending Database Technology. 2019.

[GH83] Ginsburg, S.; Hull, R.: Order dependency in the relational model. Theoretical computer science 26/1-2, pp. 149–195, 1983.

[LN16] Langer, P.; Naumann, F.: Efficient order dependency detection. The VLDB Journal—The International Journal on Very Large Data Bases 25/2, pp. 223–241, 2016.

[SGG12] Szlichta, J.; Godfrey, P.; Gryz, J.: Fundamentals of order dependencies. Proceedings of the VLDB Endowment 5/11, pp. 1220–1231, 2012.

[Sz17] Szlichta, J.; Godfrey, P.; Golab, L.; Kargar, M.; Srivastava, D.: Effective and complete discovery of order dependencies via set-based axiomatization. Proceedings of the VLDB Endowment 10/7, pp. 721–732, 2017.