"Confluence: Speeding Up Iterative Distributed Operations by Key-dependency-aware Partitioning"

# Response to Review Comments

In the previous draft, we modeled iterative distributed operations by constructing the key dependency graph (KDG) and then applied Confluence key partitioning (CKP) by finding the pure Confluence subgraph, which seemed to be a complex procedure. This intricate model is the root cause of wonders:
(a) how efficient the procedure itself is, as it seems we need to make efforts and spend time going through this procedure;
(b) what actual effect of CKP is when applied in a distributed application, as it is not directly deducible from the KDG;
(c) how CKP can be applicable to complicated distributed applications since KDG would likely be very complicated then, let alone finding out the pure Confluence subgraph.

We set off to find an easier alternative for the complicated model. In this revised version, we formalize data relations between different iterations by a much simpler notion of "key dependency". We find that the key dependency model is not only much simpler in form, but also can easily answer the above questions.

A key dependency looks like: $k \Rightarrow_{pr} f(k)$

The key dependency captures the transformation logic of distributed operations. Compared to the method based on KDG, the difference by using key dependency is:

(1) Finding the key dependency need not consider every edge between key nodes in the graph. The programmer needs only to find one or a few key dependencies that are important. In a sense, the programmer does not have to construct the whole KDG, but only some edges in the form of key dependency, which is enough for applying CKP. This change answers question (a).
(2) A key dependency not only indicates two keys are related, but also includes a parameter "pr", called dependency probability, that more precisely captures the statistical relation. With the dependency probability, the programmer can easily deduce the benefit of binding key partitions based on the key dependency. This change answers question (b).
(3) The technique of binding key partitions perfectly match the notion of key dependency. The key dependency is useful because we find that by binding the partitions of key k and key f(k) on both side of a key dependency, the shuffle size almost always decreases when compared to the random partitioning. Besides, the key dependency directly maps to the implementation. Programmers can apply CKP in their program using as little as a single line of code based on the key dependency. This change answers question (c).

We replace the KDG model with the key dependency model in the revised draft. With the new key dependency model, we find the description of applying CKP becomes tidier and clearer, because of the precise nature of key dependency. In addition, we have made lots of other changes, which are listed under Summary of Changes. They answer most of the reviewers' questions.

## Major Responses:

(1) What are the limitations of the proposed mechanism?

***Response:*** We address this question in Section 4.5, by stating those circumstances in which CKP will exhibit a limitation, and suggest solutions to these circumstances.

(2) How hard to derive a KDG, especially developers may modify the original distributed operations?

***Response:*** We address this question in Sections 3.2 and 4.5. With the new key dependency notion, the key dependency can be derive easily in most applications, with as low as O(1) effort in some cases. Besides, applying CKP based on the key dependency in the program code is simple via the ConfluencePartitioner interface, as discussed in Section 5.2.

(3) Given a developer-provided KDG, the following steps, i.e., extracting Confluence Subgraph and generating the CKP scheme, need to be automated. This part is done manually and not clearly described in the manuscript.

***Response:*** The new key dependency notion is simpler than the KDG of the previous draft, and there is no need to extract a subgraph. Though the key dependency is still manually provided by the programmer, it is much easier to understand and to derive. This is addressed in Section 3.2.

(4) The end-to-end performance needs to be evaluated to demonstrate the effectiveness of the approach.

***Response:*** We evaluate the end-to-end performance via a job's completion time. Results show that CKP can reduce the job's completion time by as much as 23%.

(5) Discussing the related work.

***Response:*** We discuss the related work of "SCOPE" in Section 7, and mention and contrast related shuffle evaluation work in the Introduction.

## Summary of changes:

1. Introduction: we stress the power of key dependency as a main contribution, and emphasize more on the effect and applicability of CKP.

2. In the background part, in order better analyze the shuffle behavior in iterative distributed operations and to avoid confusion with the map-and-reduce primitive, we redefine the iterative distributed operation model by the transform-and-shuffle primitive (Section 2.1).

3. In Section 2.2, we revise the iterative shuffle size model based on the new transform-and-shuffle primitive.

4. In lieu of the KDG proposed in the previous draft, we formally define the key dependency notion in Section 2.3.

5. We added Section 2.4 to show how the shuffle size of the random key partition scheme can be obtained from the key dependency information.

6. In the previous draft, we localized data corresponding to a pure confluence subgraph in a node. In this revised draft, we propose a similar but simpler technique called "key partition binding", in Section 3.1. The benefit of binding key partition based on the key dependency is obvious and can be easily deduced. This technique along with the notion of key dependency is the foundation of CKP.

7. Rather than merely presenting the algorithm of CKP and calculating the shuffle size as in the previous draft, we discuss and explore more in detail the property, the effect and the potential difficulty of applying CKP (Section 3.2). We discuss the complexity of applying CKP and show that it is easy with almost no cost. We discuss the features of the algorithm of CKP to clarify potential difficulties in practice. We discuss points to note in order to make CKP more effective in distributed applications.

8. In the workload skewness section, rather than to calculate the value of workload skewness iteration by iteration, which is hard to comprehend, we discuss the conditions for CKP not to increase workload skewness. Once the conditions are satisfied, CKP will not introduce extra workload skewness.

9. We re-discuss the impact of inaccurate modeling of key dependency. The notion of key dependency makes the discussion clearer.

10. In the application section, besides adjusting the description of the existing distributed applications to fit the new key dependency notion, we make two major changes. The first is that we added the PageRank and the GoogleAlbum examples (Section 4.2), which are representatives of the "join" operation. The second is that we added a subsection that discusses limitations when applying CKP in distributed applications and suggests solutions or workarounds to solve these problems (Section 4.5).

11. We remove the map-side combine part in the previous draft, because we found that after using the transform-and-shuffle primitive, the map-side combine is naturally a part of the transform operation and need not be explicitly mentioned.

12. In the implementation section (Section 5.2), we show the usage of the ConfluencePartitioner interface by comparing the code after applying CKP in the program. The code comparison shows that the ConfluencePartitioner interface is convenient and applying CKP is very easy.

13. We improve the evaluation (Section 6) by adding the completion time evaluation to verify the end-to-end performance benefits of CKP.

14. In related work (Section 7), we added a discussion on similar work on shuffle partitioning based on data relations between iterations.