

HiCoMo: High Commit Mobile Transactions

Minsoo Lee and Sumi Helal

Dept. of Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611-6120

{mslee, helal}@cise.ufl.edu

Technical Report # 99020

Abstract

We introduce a new mobile transaction model applicable to decision-making applications over aggregate data warehoused on mobile hosts. The model allows the aggregate data to be updated in disconnection mode, while guaranteeing a very high rate of commitment on reconnection. We name such transactions High Commit Mobile Transactions, or HiCoMo. At reconnection-time, HiCoMo's are analyzed and several base (fixed network) transactions are generated in order to bring the same effect upon the base tables from which the aggregates are derived. In this Technical Report, we provide a formal definition for the concepts related to HiCoMo transactions, and a transformation algorithm that is used to analyze them and generate base transactions. We discuss issues of Satisfiability (does there exist such a base transaction configuration?) and Computability (can a satisfying base transaction configuration be computed within some time limits and acceptable error margin?) Simple example scenarios are provided to demonstrate the usefulness of our approach.

1. Introduction to HiCoMo Transactions

- Environment

HiCoMo's are used in an environment where there exist base tables on nodes (i.e., base nodes) in the fixed network and a data warehouse - an aggregate of the data in the base tables - existing on mobile hosts. The mobile host will be mostly disconnected from the fixed network.

- Transaction Semantics

Transactions will be initiated from the fixed network on the base tables, which are called source transactions. Transactions may also be initiated on the mobile hosts performing operations on the aggregate data. The transactions that are initiated on the mobile host are called *HiCoMo transactions*. HiCoMo's are issued on statistical data and can consist of updates, and although the mobile host can possibly be disconnected from the fixed network, the transaction is permitted to run on the local data warehouse. The updates performed on the data warehouse should be reflected later on upon the base tables. When the mobile host reconnects to the fixed network, HiCoMo transactions are processed and transformed into updates on the base tables. It may not be possible to

generate updates on the base tables that can accurately reflect the effects of the HiCoMo transactions on the data warehouse. In this case, a reasonable error margin is allowed. Therefore, it is almost always guaranteed that the HiCoMo's will succeed.

2. Overview of Problem and Approach

2.1 Problem

The key point in HiCoMo transactions is the processing carried out at the time when the mobile host reconnects to the fixed network and generates source transactions to create the effect that was done by a HiCoMo transaction on the mobile host. There are various ways that these source transactions can be generated. As these transactions will be performed on a large set of data, the operations can be spread upon several of the base tables and the operations can help each other to bring the overall effect of a HiCoMo transaction. If the overall effect cannot be brought upon by any of the configuration of the generated transactions, an alternate method of generating transactions can be done by allowing an error margin between the actual updates on the base tables and the updates on the data warehouse. The following issues are to be resolved :

- How should the source transactions, which have the overall effect of a HiCoMo transaction, be generated?
- How can error margins be considered in generating these source transactions?
- What transaction model may be applied, or may be modified to adapt to this situation?
- What kind of acceptance criteria - a criteria that is used to solve conflicts that occur among different HiCoMo transactions - may be reasonable to apply to this situation?

2.2 Approach

- Assumptions

(1) The aggregate data that will be stored in the data warehouse is one of the following : average, summation, minimum, maximum,

(2) Only commutative operations will be applied to the data warehouse such as : addition(+), subtraction(-). The commutative property is important because without this property a lot of conflicts will occur among different HiCoMo transactions applied on

the average by 3000 and has an error margin of 20, the actual update effect on the base tables may be an increase within the range of 2980 to 3020

- Generation of source transactions

The source transactions will be generated based on the combinations of the aggregate type (i.e., average, summation, minimum, maximum), the operation type (i.e., addition, subtraction), and the base table configuration.

Satisfiability (does there exist such a source transaction configuration?) and Computability (can a satisfying source transaction configuration be computed within some time limitation?) issues should also be considered.

Another alternative approach currently possibly applicable is to reinforce the HiCoMo transaction effects on the base tables without reapplying the entire transaction logic. This is based on the idea of just bringing the effect up to date on the base tables.

- Error margin consideration

The error margin could be initially disregarded to maintain consistent data among the base tables and the data warehouse. After the consistent source transaction combinations are considered and none of them are possible candidates due to conflicts with other HiCoMo transactions or base transactions performed during disconnection, the error margin is included into the source transaction generation. The margin may start from a small amount to an increasing amount. The increase in the error margin may be derived from the hints obtained from previous trial and errors of trying to generate source transactions.

- Use an extended nested transaction model

The generated source transactions will be applied as a whole to the base tables. If some of the source transactions abort, another transaction configuration should be tried.

A transaction model which can support a concurrent and yet individual abort scheme should be adopted. The nested transaction model can allow this kind of behavior with concurrent and independent subtransactions. Each of the source transactions can be modeled as a subtransaction. Although this model is sufficient to apply a single source transaction configuration on the base tables, it does not capture the semantics of retrying another source transaction configuration when some subtransactions abort. Thus, the nested transaction semantics need to be modified with the semantics of retrying a subtransaction that has aborted using a new subtransaction.

This will modify the commit semantics of the nested transaction model.

- Use Convergence as an acceptance criteria

Convergence is a simple yet very widely recognized acceptance criteria. It basically states that if eventually the base tables state is equal to what is reflected on the data

warehouse (i.e., converges), it satisfies convergence. If the operations are commutative, this acceptance criteria can be mostly satisfied. But, it is still possible that the operations are commutative and the acceptance criteria may not be satisfied due to interactions with other HiCoMo transactions. Some other HiCoMo transaction may have updated the base tables, and although the source transactions that were generated are applied to the base tables, it may not bring them to a state where the base tables state and the data warehouse state are equivalent.

This simple convergence criteria need to be investigated with respect to the HiCoMo transactions.

3. Formal Definition of HiCoMo Transactions and Problem

In order to clarify the semantics of a HiCoMo transaction, a formal definition is provided in this section. The problem to be tackled in supporting HiCoMo transactions is also identified with a formal representation through this process of formalization. Several preliminary definitions are needed.

Definition 1. An **Aggregate Function (AF)** is a function that is applied to several Base Tables to obtain a table of aggregate values that represent statistics of the Base Tables. It is defined as a tuple of the following form :

$$AF = (AGG_OP, AGG_FIELD, GROUP_FIELD, n, \{SP_1, SP_2, SP_3, \dots, SP_n\})$$

- **AGG_OP** is an aggregate operation such as average(AVG), summation(SUM), minimum(MIN), maximum(MAX) etc.
- **AGG_FIELD** is the field(column) name of the final table that the AGG_OP should be applied to.
- **GROUP_FIELD** is the field(column) name of the final table that a group-by operation should be applied when performing the AGG_OP on AGG_FIELD.
- **n** is the number of Base Tables that are expected as input to the AF function. (i.e, 10, 25 ...)
- **{SP₁, SP₂, SP₃, ..., SP_n}** is the set of select-project operations that are to be performed on each of the Base Tables. Each SP_i is the select-project operation that is to be performed on the Base Table **B_i** (given as the input to AF) where $1 \leq i \leq n$.

When the AF is applied to a set of Base Tables, it yields a single table containing aggregate values.

$$\begin{aligned} AF(\{B_1, B_2, B_3, \dots, B_n\}) \\ &= AGG_OP(AGG_FIELD, GROUP_FIELD, (\bigcup_{i=1, \dots, n} SP_i (B_i))) \\ &= AGG_OP(AGG_FIELD, GROUP_FIELD, T1) \\ &= AT \end{aligned}$$

- T1 is the table that is the result of calculating $\bigcup_{i=1, \dots, n} SP_i (B_i)$.

- **AT** is the resulting Aggregate Table.
- Note that these input Base Tables can have a different schema, but after the select-project (SP_i) operations are performed on each of the tables, the resulting tables should be in the same form that commonly allows an aggregation operation on a specific field along with a group-by operation.

Definition 2. A tuple (**AF**, **B_SET**) is called the **Environment Tuple**, and represents the current environment that a HiCoMo transaction is used.

- **AF** is the Aggregate Function that maps a set of n Base Tables to a single Aggregate Table (i.e., the domain of the function is : $T^n \rightarrow T$, where T is a table).
- **B_SET** is a set {B₁, B₂, B₃, ..., B_n }, where each B_i is a Base Table existing in the fixed network. Actually, the set of Base Tables can also be viewed as one

- $\{ B_1, B_2, \dots, B_n \}$ is the set of Base Tables reflecting the state *before* the HiCoMo Transaction HTR is transformed and applied to the base tables.
- AT is the Aggregate Table *before* the HiCoMo Transaction HTR is applied.
- $\{ IC_1, IC_2, \dots, IC_j \}$ is the set of Integrity Constraints on the Base Tables.
- $\{ HTR_1, HTR_2, \dots, HTR_m \}$ is the set of *possibly* conflicting HiCoMo Transactions with HTR *before* HTR is transformed and applied. In our case, we only consider commutative operations for HiCoMo Transactions. Thus, HiCoMo Transactions do not conflict with each other.
- $\{ BTR_1, BTR_2, \dots, BTR_p \}$ is the set of conflicting Base Transactions with HTR.
- $\{ BTR_{p+1}, BTR_{p+2}, \dots, BTR_k \}$ is the set of Base Transactions that were generated by the HiCoMo Transaction HTR. This may also be a set containing only ABORT, which means that HTR cannot be transformed and must be aborted.
- $\{ B'_1, B'_2, \dots, B'_n \}$ is the set of Base Tables reflecting the state *after* the HiCoMo Transaction HTR is transformed and applied to the base tables.
- AT' is the Aggregate Table *after* the HiCoMo Transaction is applied.
- $\{ HTR'_1, HTR'_2, \dots, HTR'_m \}$ is the set of *possibly* conflicting HiCoMo Transactions with HTR *after* HTR is transformed and applied. In our case, as there are no conflicts among HiCoMo Transactions, it should be the same as *before* the HiCoMo Transaction STR is applied.

The following sections focus on how to provide this HiCoMo Transformation Function. An algorithm for performing the transformation and an extended nested transaction model are developed to support the HiCoMo Transaction concept.

4. Algorithm for Generating Base Transactions

The Transaction Transformation Function, which was previously described, is the key focus of this section. Recall that the TTF needs as input the HiCoMo Transaction, the Aggregate Function, the Base Tables, the Aggregate Table, the Integrity Constraints, the conflicting HiCoMo Transactions, and the conflicting Base Transactions. All of these factors are considered when transforming a HiCoMo Transaction into a set of Base Transactions.

A general description of the algorithm for performing the transformation is as follows.

1) Conflict Detection : Detect a conflict between the given HiCoMo Transaction and the other HiCoMo Transactions that are not yet transformed into Base Transactions, and also the conflicting Base Transactions. In our case, actually no conflicts are detected among HiCoMo Transactions. If a conflict is detected, abort the given HiCoMo Transaction that is being considered for transformation. This simple abort strategy is due to the fact that we want to provide durability for Base Transactions, and more significantly, there is no way to control what happened within other Base Transactions.

2) Initial Base Transaction Generation : If there is no conflict, decide what kind of initial Base Transactions should be created. This is based on the information provided by

the HiCoMo Transaction and other factors such as the Aggregate Function and Base Tables. Once these are decided, run these transactions as sub-transactions of an extended nested transaction.

3) Sub-transaction Abort and Alternate Base Transaction Generation : Some of the sub-transactions may abort due to integrity constraints that restrict certain updates from being performed. This case is difficult to predict prior to generating the Base Transaction. Therefore, the aborted transactions need to be dealt with afterwards. The aborted sub-transactions will affect the overall result of the HiCoMo Transaction, and need to be somehow compensated for. If the difference that the aborted sub-transactions create is within an error margin, it is allowed to just simply finish with several aborted sub-transactions. Otherwise, the update amount for the aborted sub-transactions may be redistributed among themselves and retried. If after a certain amount of redistribution is done and the abort of the sub-transactions are still occurring, the error margin may also be considered in the redistribution. If the sub-transactions can succeed at some point, the transformation is complete. Otherwise, the HiCoMo Transaction need to be aborted.

A detail description of each of these steps are provided in the following sections.

4.1 Conflict Detection

One of the assumptions we made earlier is that we only consider commutative operations for HiCoMo Transactions. In other words, the OP part of the HiCoMo Transaction will only be allowed to be addition (+) or subtraction (-), and not division nor multiplication. This assumption makes it possible to reorder the HiCoMo Transactions, and eliminates the conflicts among HiCoMo Transactions.

Therefore, we only need to detect conflicts between the given HiCoMo Transaction and the Base Transactions. The Base Transactions can be composed of any type of operations including not only addition or subtraction but also division or multiplication. Thus, Base Transactions are not commutative with HiCoMo Transactions. So, the *order between the HiCoMo Transaction and Base Transactions are important*. We also want to guarantee durability for Base Transactions, as they are always performed in the fixed network without disconnection problems. This means that when there is a conflict between a HiCoMo Transaction and some Base Transactions, *the HiCoMo Transaction should be aborted* and not the Base Transactions.

The conflict detection can be implemented by a *time-stamp based optimistic concurrency control strategy* [3,4]. In this strategy, a transaction goes through three phases : execution, validation, and update. The execution phase in our case is done when the HiCoMo Transaction performs updates to the aggregate table on the mobile host. This phase is actually considered as starting from the disconnection point (i.e., all read operations are done at this point) and ends when the mobile host is reconnected. The validation phase checks for any conflicts of these updates with any other updates. If the updates performed by the HiCoMo Transaction are younger (i.e., more recent) than any other conflicting Base Transaction updates, then the HiCoMo Transaction updates are allowed to be actually applied on the Base Tables. Otherwise, the HiCoMo Transaction is

aborted. The validation phase starts when the mobile host has reconnected to the fixed network. The update phase is done by transforming the HiCoMo Transaction into Base Transactions and actually applying them to the Base Tables. Using this strategy, the transactions will be serialized in time-stamp order.

Note that the HiCoMo Transaction will eventually need to update a subset of the tuples in the Base Tables that are aggregated in the table of the mobile host. This subset may be large and may possibly result in many conflicts, but the basic problem is that there is no control over what amount of updates are performed by the Base Transactions. Thus, it is difficult to enforce a desired update effect by the HiCoMo Transaction. Advanced issues and solutions related to this problem are discussed later on.

4.2 Initial Base Transaction Generation

Recall that a HiCoMo Transaction is composed of {SEL, GROUP_FIELD, OP, AGG_FIELD, VAL, ERR, AT}. A simple example of a HiCoMo Transaction is as follows.

- SEL : manager
- GROUP_FIELD : Level
- OP : increase (+)
- AGG_FIELD : Salary
- VAL : 2000
- ERR : 100
- AT : aggregate table generated by the SQL query

```
SELECT Level, AVG(Salary)
FROM Employee
WHERE Employed_date < 01/01/1995
GROUPBY Level
```

This HiCoMo Transaction will increase the average Salary of the manager Level employees who have started working prior to the date 01/01/1995 by \$2000 with an error margin of \$100.

By carefully looking into this example, the relationship between the HiCoMo Transaction and the desired generated Base Transactions can be observed.

First, the *target tuples of the generated Base Transactions* are decided based on : (1) which Base Table tuples have been aggregated into the aggregate table (i.e., WHERE clause of SQL query that produces AT), (2) and also which aggregates are selected by the HiCoMo Transaction within the aggregate table (i.e, SEL, GROUP_FIELD). If we denote the selection condition of (1) as S, then we obtain the following query that results in the target tuples of the generated Base Transaction.

```
SELECT *
FROM base tables
WHERE S AND GROUP_FIELD = SEL
```

In the case of the above example, it would result in :

```
SELECT *
FROM Employee
WHERE Employed_date < 01/01/1995 AND Level = manager
```

Second, the *different types of aggregate operations (i.e., AGG_OP in the aggregate function : AVG, SUM, MIN, MAX) used when creating the aggregate table* must be considered to generate the correct Base Transactions. The following rules are applied to perform the transformation. They use the AGG_OP, and the OP, AGG_FIELD, VAL information of the HiCoMo Transaction.

- If AGG_OP = AVG, generate Base Transaction(s) that will update each of the target tuples by applying the OP on the AGG_FIELD with the amount of VAL. In the above example, this means that the Base Transaction(s) will increase each of the Salary fields of the target tuples by \$2000. This should bring about the effect of increasing the average by \$2000.
- If AGG_OP=SUM, generate Base Transaction(s) that will update the first tuple among the target tuples by applying the OP on the AGG_FIELD with the amount of VAL. If we assume that the aggregate table in the above example was generated by a SUM, this means that the Base Transaction(s) will increase the Salary field of the first target tuple by \$2000. This should bring about the effect of increasing the summation by \$2000.
- If AGG_OP=MIN and OP=subtraction(-), generate Base Transaction(s) that will update the target tuple with the minimum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL. If we assume that the aggregate table in the above example was generated by a MIN, this means that the Base Transaction(s) will decrease the Salary field of the minimum target tuple by \$2000. This should bring about the effect of decreasing the minimum by \$2000.
- If AGG_OP=MIN and OP=addition(+), generate Base Transaction(s) that will update the target tuple with the minimum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL. Also, other Base Transaction(s) that update target tuples falling below this new minimum are generated. If we assume that the aggregate table in the above example was generated by a MIN, this means that the Base Transaction(s) will increase the Salary field of the minimum target tuple by \$2000. Additional Base Transaction(s) are generated to update other target tuples that have a Salary field below this new minimum to exactly the minimum value. This should bring about the effect of increasing the minimum by \$2000.
- If AGG_OP=MAX and OP=addition(+), generate Base Transaction(s) that will update the target tuple with the maximum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL. If we assume that the aggregate table in the above example was generated by a MAX, this means that the Base Transaction(s) will increase the Salary field of the maximum target tuple by \$2000. This should bring about the effect of increasing the maximum by \$2000.
- If AGG_OP=MAX and OP=subtraction(-), generate Base Transaction(s) that will update the target tuple with the maximum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL. Also, other Base Transaction(s) that update target tuples exceeding this new maximum are generated. If we assume

that the aggregate table in the above example was generated by a MAX, this means that the Base Transaction(s) will decrease the Salary field of the maximum target tuple by \$2000. Additional Base Transaction(s) are generated to update other target tuples that have a Salary field exceeding this new maximum to exactly the maximum value. This should bring about the effect of decreasing the maximum by \$2000.

We assume that one Base Transaction is generated per Base Table. Therefore, if multiple Base Tables are aggregated, multiple Base Transactions may be generated by the HiCoMo Transaction. There are transaction granularity issues related to this decision of a per Base Table transaction generation. If the Base Transaction granularity is too small such as being generated for each target tuple rather than each Base Table, the problem is that the processing overhead of these large number of generated transactions may be tremendous. But the good thing is that each failure of a Base Transaction is localized. On the other hand, if the Base Transaction granularity is too large and a single target tuple update fails, then the whole Base Transaction will need to be reconsidered. This decreases the transaction processing overhead due to the small number of Base Transactions generated, but can create unnecessary aborting of even succeeded large portions of the transaction.

There are also alternate ways of generating the Base Transaction(s) by using a different distribution of the update values among the target tuples of the Base Table(s). A simple example for the SUM discussed above, is to distribute parts of the total update amount among all of the target tuples instead of just the first one. The goal of the initial generation of the Base Transaction(s) is to not only *allow as uniform updates among the tuples as possible* but also to minimize the generated Base Transaction life time by *updating as less target tuples as possible*.

4.3 Sub-transaction Abort and Alternate Base Transaction Generation

The initial Base Transaction(s) generated are executed as sub-transactions of an extended nested transaction that will be explained in the next section.

Although we are guaranteed that the Base Transactions generated will not conflict with other Base Transactions, it is still possible for the generated Base Transactions to be aborted. One of the main reasons would be the existence of *integrity constraints* on the fields that are being updated. An example of such an integrity constraint would be that a field should have a non-negative value where the Base Transaction attempts to drive the value below zero. The sub-transaction that this Base Transaction is being executed in will be aborted.

The extended nested transaction localizes failure of sub-transactions. Therefore, individual aborts of sub-transactions do not directly affect the execution of other sub-transactions, but will definitely affect the overall aggregate update desired by the HiCoMo Transaction. The strategy proposed for this situation is :

- If the aborting of the sub-transactions still leave the overall update effect within a pre-specified error margin of the HiCoMo Transaction, it may be alright to leave the aborted sub-transactions as they are.
- If a large number of sub-transactions abort and the overall update effect is not within the pre-specified error margin of the HiCoMo Transaction, a redistribution or alternate strategy of generating Base Transactions for the aborted sub-transactions should be activated.

The error of the overall update effect can be calculated using the total number of target tuples, the number of aborted sub-transactions, the type of aggregate, the HiCoMo Transaction operation and value, the value of the target tuple before the sub-transaction aborts, and the value of the target tuple after the sub-transaction succeeds.

When aborted sub-transactions need to be rerun, they will be generated according to the following rules.

- If $AGG_OP = AVG$, regenerate Base Transaction(s) for the aborted sub-transactions with consideration of error margins. The error margin that should be applied depends on the total abort ratio that had occurred. Assuming 10% of the sub-transactions had aborted, we apply 10% of the error margin to the update values on each of the aborted sub-transactions. Thus, the VAL will be decreased by 10% of the error margin and applied to each target tuple of the aborted sub-transactions. In the above example, this means that the Base Transaction(s) will increase each of the Salary fields of the target tuples by only \$1990 ($=\$2000 - \$100 * 10\%$). This should bring about the effect of increasing the average by \$2000 with an error margin within \$10 ($=\$100 * 10\%$). The error margin would then be applied in the next round with double(20%), triple(30%) the initial ratio as this process goes on until the error margin is no longer adjustable. Each time before the regeneration process of the Base Transactions, the total error is calculated including the aborted sub-transactions. If the result is not satisfactory, another round of regeneration starts. Otherwise, it will stop successfully. After going through all of the regeneration, it may be possible that the total error margin cannot be satisfied. In this case, the HiCoMo Transaction is aborted. A simple reason why redistributing values was not used before trying to use the error margin was that the aborted sub-transactions were *all* already making the resulting values to exceed or go below some integrity constraint. Trying to even out the values among the individual target tuples of the aborted sub-transactions will still make them *all* to abort. The only way to save some of the sub-transactions is to increase (or decrease) the potentially good ones by just a small value and put all of the left over big burdens on the potentially bad ones. This will create a hazard for the error margin and may become uncontrollable. Thus, we avoid this approach.
- If $AGG_OP = SUM$, regenerate Base Transaction(s) that will update the *next* tuple among the target tuples by applying the OP on the AGG_FIELD with the amount of VAL. If we assume that the aggregate table in the above example was generated by a SUM, this means that the Base Transaction(s) will increase the Salary field of the *next* target tuple by \$2000. This should bring about the effect of increasing the summation by \$2000. Retry this for 10% (a fixed number) of the total tuples. If it

doesn't work for this number of retries, redistribute the SUM amount in an evenly fashion over all of the target tuples. In other words, apply $(VAL/\text{number of target tuples})$ on each of the tuples. If an abort of a sub-transaction occurs, proceed as in the above AGG_OP=AVG case where error margins are considered.

- If AGG_OP=MIN and OP=subtraction(-), regenerate Base Transaction(s) that will update the target tuple with the minimum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL adjusted by multiples of 10% of the error margin. Each time, the sub-transaction is aborted, the error margin increases(i.e., 20%, 30%, ...). If the error margin cannot be satisfied after all retries, then abort the HiCoMo Transaction. If we assume that the aggregate table in the above example was generated by a MIN, this means that the Base Transaction(s) will decrease the Salary field of the minimum target tuple by \$1990 ($\$2000 - \$100 * 10\%$). This should bring about the effect of decreasing the minimum by \$2000 with an error margin of 10%.
- If AGG_OP=MIN and OP=addition(+), regenerate Base Transaction(s) that will update the target tuple with the minimum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL adjusted by multiples of 10% of the error margin. Also, other Base Transaction(s) that update target tuples falling below this new minimum are generated. Each time, any sub-transaction is aborted, the error margin increases(i.e., 20%, 30%, ...). If the error margin cannot be satisfied after all retries, then abort the HiCoMo Transaction. If we assume that the aggregate table in the above example was generated by a MIN, this means that the Base Transaction(s) will increase the Salary field of the minimum target tuple by \$1990 ($\$2000 - \$100 * 10\%$). Additional Base Transaction(s) are generated to update other target tuples that have a Salary field below this new minimum to exactly the minimum value. This should bring about the effect of increasing the minimum by \$2000 with an error margin of 10%.
- If AGG_OP=MAX and OP=addition(+), regenerate Base Transaction(s) that will update the target tuple with the maximum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL adjusted by multiples of 10% of the error margin. Each time, the sub-transaction is aborted, the error margin increases(i.e., 20%, 30%, ...). If the error margin cannot be satisfied after all retries, then abort the HiCoMo Transaction. If we assume that the aggregate table in the above example was generated by a MAX, this means that the Base Transaction(s) will increase the Salary field of the maximum target tuple by \$1990 ($\$2000 - \$100 * 10\%$). This should bring about the effect of increasing the maximum by \$2000 with an error margin of 10%.
- If AGG_OP=MAX and OP=subtraction(-), regenerate Base Transaction(s) that will update the target tuple with the maximum value on the AGG_FIELD by applying the OP on the AGG_FIELD with the amount of VAL adjusted by multiples of 10% of the error margin. Also, other Base Transaction(s) that update target tuples exceeding this new maximum are generated. Each time, any sub-transaction is aborted, the error margin increases(i.e., 20%, 30%, ...). If the error margin cannot be satisfied after all retries, then abort the HiCoMo Transaction. If we assume that the aggregate table in the above example was generated by a MAX, this means that the Base Transaction(s) will decrease the Salary field of the maximum target tuple by \$1990 ($\$2000 -$

\$100*10%). Additional Base Transaction(s) are generated to update other target tuples that have a Salary field exceeding this new maximum to exactly the maximum value. This should bring about the effect of decreasing the maximum by \$2000 with an error margin of 10%.

5. Extended Nested Transactions

The Extended Nested Transaction model is a simple extension to the traditional nested transaction model. The main feature is that it has a modified commit semantics. The failure of the sub-transactions are localized, but the failures are not just left as it is. When a sub-transaction fails, the top-level transaction is notified of the failure. When all of the sub-transaction fates are decided, the top-level transaction will go on to the next step which is a modified step added by our model. The top-level transaction will readjust the sub-transaction parameters and rerun the sub-transaction. If the sub-transaction aborts again, it may be rerun again with another parameter settings. The criteria for stopping this retrying semantics is decided by the algorithm discussed above. An aggregate update amount should be reflected within a pre-specified error margin given to the HiCoMo Transaction. Therefore the condition for committing the top-level transaction is as follows.

```
If fate of all sub-transactions are decided
{
    Gather all results of sub-transaction.
    If results satisfy error margin of HiCoMo Transaction, commit.
    If not,
    {
        If error margin has been fully exploited, abort.
        If not, use regeneration algorithm of Base Transactions to
        retry sub-transactions.
    }
}
```

6. Example Scenario using HiCoMo Transactions

HiCoMo Transactions are based on a new type of applications that may be very common in the mobile computing environment. A CEO of a company may want to carry around only HiCoMo data of his company rather than the *whole* database which may be huge. He is the person who can make company-wise decisions and can directly update the HiCoMo data that he carries around and will be later on reflected onto the real database of his company.

A simple example scenario is provided in this section to demonstrate how the concept works.

Suppose there are the following two tables Employee1 and Employee2 in the company. These tables are separately managed on different sites on the fixed network, but have the same schema (for simplicity purpose).

Employee1 on site A

Emp_no	Name	Level	Salary	Dept
104467	Jack Crane	Manager	79000	Management
350933	Chris White	Programmer	68000	Design
230988	Smith Gordon	Manager	69000	Manufacture

Employee2 on site B

Emp_no	Name	Level	Salary	Dept
308867	Janette Sanders	Programmer	66000	Design
111436	Sue Hill	Programmer	74000	Manufacture
566217	Bart Simpson	Manager	82000	Management

The CEO takes the aggregate table which consists of the average salary of his 6 (this is just a simplified version of the company) employees grouped by the category of Level on his mobile host.

The table should look like this.

Level	AVG (Salary)
Programmer	69333
Manager	76666

The CEO travels around and finds out that his programmers are being payed very little compared to other competing companies and wants to increase the average salary of the programmers so that these important human resources don't leak out of his company.

The CEO wants to increase the average by \$6000 with an error margin of \$100.

But there is an integrity constraint on the Salary field such that the Salary of an employee who is not at the Manager Level should not exceed \$80000.

When the CEO updates the value on his mobile computer, the Salary for the Programmer should change to 75333 [+/-100].

The CEO later reconnects to the fixed network and downloads his transactions on the company database. The base transactions will be generated from the HiCoMo transaction according to the algorithms discussed in earlier sections.

Assuming that there were no conflicting base transactions with the HiCoMo transactions, we go to the next step where the HiCoMo transaction is transformed into base transactions.

Two base transactions are identified, one for each Employee table. The two base transactions will attempt to increase the Salary of the Programmers by \$6000. These two base transactions are executed as sub-transactions of an extended nested transaction. The updates on Employee1 is successful, but the sub-transaction executed on Employee2 is aborted because of the integrity constraint enforced on Sue Hill (her salary will exceed

\$8000). The sub-transaction on Employee2 is readjusted to increase only by \$5990 (= \$6000 - \$100 * 10%). This time the sub-transaction succeeds. Thus, the actual error margin is within \$10 (= \$100 * 10%).

The resulting Employee tables will look like the following.

Employee1 on site A

Emp_no	Name	Level	Salary	Dept
104467	Jack Crane	Manager	79000	Management
350933	Chris White	Programmer	74000	Design
230988	Smith Gordon	Manager	69000	Manufacture

Employee2 on site B

Emp_no	Name	Level	Salary	Dept
308867	Janette Sanders	Programmer	71990	Design
111436	Sue Hill	Programmer	79990	Manufacture
566217	Bart Simpson	Manager	82000	Management

The resulting CEO table should also be updated to capture the correct average values. It should look like the following.

Level	AVG (Salary)
Programmer	75326
Manager	76666

An error margin of \$7 (= \$75333 - \$75326)! This is definitely within the error margin of \$100.

7. Further Improvements and Issues

There are several issues that can further improve the proposed approach.

First, the *partitioning granularities of the base table* will affect the performance of the algorithm. If the base tables are not partitioned, then in our case only a single base transaction will be generated from the HiCoMo transaction. This will create the danger of a high abort rate due to the large granularity of the sub-transaction. Therefore, tuple level sub-transaction generation schemes may also be investigated. But it should be well understood that the finer the granularity, the more sub-transactions are generated. The large number of generated sub-transactions will create again a large transaction processing overhead. The granularity tradeoff is an important factor in the performance of the algorithm.

Second, conflicts between base transactions and HiCoMo transactions may be reduced by employing some advanced mechanisms which have not yet been fully explored in our work. Adding *vertical partitioning* may help resolve conflicts between base transactions and HiCoMo transactions if they modify different fields. Another way is that although the conflicts may exist, the HiCoMo transaction may not just abort but take into consideration its own updates together with the updates performed by the base

transactions and incorporate it into the aggregate update value. This may although need special logging of the base transaction value updates and complicates the consistency management. Because actually there is no control over what updates a base transaction had performed.

Third, merging of histories may be provided instead of generating transactions on the base tables. This would be beneficial if a large number of aggregate copies of the base tables are carried around and need to reduce the transaction processing overhead at the base tables. But, we assume right now that this kind of summary data is mostly carried around by only a few executives of companies etc.

8. Conclusion

In this report, we have introduced a new type of transactions that is expected to be very useful in the mobile computing environment, called HiCoMo Transactions. HiCoMo Transactions allow for aggregate data carried on mobile hosts to be updated during disconnection and can reintegrate the update operations performed on the aggregate data into the base tables existing in the fixed network. As these updates are aggregate updates, several base transactions which can bring the same effect upon the base tables need to be generated. We have provided a formal definition for the concepts related to the HiCoMo Transactions, and have provided a transformation algorithm to support HiCoMo Transactions. Future work will focus on reducing the conflict between the base transactions and HiCoMo Transactions, and fine tuning granularities for partitioning base tables to enhance the extended nested transaction processing, and also investigating into using a merging approach of histories.

References

- [1] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In Proc. Of ACM-SIGMOD Int'l Conf. on Management of Data, p. 173-182, Montreal, Canada, 1996.
- [2] Q. Lu and M. Satyanarayanan, "Isolation-Only Transactions for Mobile Computing," Operating Systems Review, Volume 28, Number 2, 1994
- [3] R. Chow and T. Johnson, Distributed Operating Systems and Algorithms, Addison-Wesley, 1997.
- [4] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [5] M. Dunham, A. Helal, and S. Balakrishnan, "A Mobile Transaction Model that Captures Both the Data and Movement Behavior," the ACM-Baltzer Journal on Mobile Networks and Applications (MONET), Volume 2, Number 2, pp149-162, October 1997.