# Overview: A hierarchical block-oriented database

The goal of this project is to create a hierarchical block-oriented database. This database will be of use in situations in which (1) it is not critical that changes to the database be reflected immediately; (2) changes to some elements of the database necessitate changes to many other elements of the database; and (3) data is hierarchically structured, with child objects needing access to parent objects, especially where a change in a parent object may necessitate changes in thousands of child objects. The block computation database achieves these goals by keeping related data physically together as much as possible, but also allowing a messaging scheme for communication between pieces of data that are not proximate.

Block databases. An example of a block database is described here: <http://www.vldb.org/pvldb/vol6/p2014-xie.pdf>. The essence of block computation is that we store data in blocks, which can be persisted, and when a number of updates that need to be made to the block accumulate, we make all of these changes at the same time. These changes might result in new updates that are needed to other blocks. This can thus be seen as equivalent to creation of a massive spreadsheet, with changes in data in some parts of the spreadsheet (in some blocks) affecting calculations elsewhere in the spreadsheet (in other blocks). The block computation approach minimizes disk I/O, at the expense of sometimes having updates take a little while to be incorporated.

Hierarchical structure. Cells in the database can be organized into hierarchical structures, so that a cell has access to the contents of its parent cell, and so that changes to the parent cells can produce immediate changes to many child cells. Because these child cells will generally be stored in the same block or blocks, we can update the child cells based on the change in the parent cell with much less disk I/O than would be necessary with a relational database. This will enable a scenario in which we store information about tables hierarchically. A hierarchy might consist of Group of Tables, Table, Table Column, Row, Item in Cell, Result of Cell Edit. In making the calculation of a cell at the bottom of the hierarchy (say, Result of Cell Edit), information from each of the higher ranking cells can be obtained. Thus, if a change is made to information relevant to the group of tables, its child cells may be set for recalculation, and thus many can be calculated at once, without loading information from many different points of the database. If this were done on a relational database, many joins would be necessary to bring in information from tables representing each level of the conceptual hierarchy.

Messaging. Cells may also be recalculated based on changes not within the same hierarchy. A goal is to be able to support large update volumes, and to ensure that updates occur even if there are storage system failures, but not necessarily make updates cascade instantaneously. We will not make changes immediately but instead will try to group related changes. For example, suppose that many of Cells A,1..A,100000000 are dependent on some of the cells B,1..B,100. If a change is made to B,1, we might need to make changes to many thousands of cells located in a number of blocks. Rather than do each of these changes individually, we will group the changes based on the location of the blocks, and create queue messages with these grouped changes. A single queue will correspond to a group of blocks, so even if the target cells are in many blocks, only a small number of queue messages may be needed. The queues, meanwhile, will also include messages from other changes, for example, if there is a change to B,2. Thus, a large number of changes that all affect a single block, resulting both from the change in B,1 and from changes in other cells, can be processed simultaneously. A background task gets items from the queue and prioritizes which block to read and update, based on the number and importance of the changes. The justification for this approach is that hard disk read speed is generally not as significant a factor in total database performance as hard disk seek speed. So, once we are reading a section of a file, we might as well read in the entire section rather than just a piece of it and make a number of changes at once. A potential performance bottleneck may be the need to deserialize an entire block, but a potential improvement later will be to deserialize only the portions of the block that are needed.

# Cell Location and Organization

To allow for related data to be stored together, associated with each cell will be an array of LocationIdentifier objects that indicate where the hierarchical location of the cell. LocationIdentifier can be used to store an ID of type string, Guid, byte, int16, int, or long, and it must be designed in such a way that the object uses a minimum amount of space. Thus, LocationIdentifier includes a byte enum LocationIdentifierType

public enum LocationIdentifierType : byte { stringType, guidType, intType, longType };

We also define a static class MakeLoc with the following method:

public static LocationIdentifier[] From(IEnumerable<object> x)

Each location must be unique. The most general area of storage is listed first, the most specific area of storage is listed last. So, {“United States”, “Texas”, “Dallas” } would represent a cell with information about Dallas, Texas. The data structure for that cell will also allow information about Texas or about the United States. The cell value for Dallas might include various types of information about population, including information about the percent of the country’s population located in that place. Because of the hierarchical structure, when a change is made to the Dallas population, the Texas and U.S. populations can be adjusted accordingly, and also the population of the U.S. can be looked up (without further disk I/O) when calculating the percentage of the U.S. population represented by Dallas. Of course, if the U.S. population is updated, the percentage of the U.S. population represented by other cities can also be changed immediately, without separately loading those cities if they are in the same block.

Sometimes, the first LocationIdentifier is used to indicate the type of information or calculation to be performed (if we wish to group items of the same calculation type to be calculated together). So, we might have a cell structure like {“Population”, “United States”, “Texas”, “Dallas”}. This might make sense if the population data is highly interdependent and injecting other types of data (say, weather and economic data) would mean that more changes would need to be propagated to other areas. Other times, the last LocationIdentifier can be the type of calculation (if there otherwise would be multiple cells at the rest of the location). This might make sense if it is useful to store the population and weather data for Dallas together (for example, because we calculate some value that involves both). At still others, the type of calculation can be omitted, if there is only one type of object for the hierarchy.

As a result of this approach, we can use a nested binary tree structure to figure out where a particular item is or should be stored. There is an outer binary tree and an inner binary tree. The binary tree information is described in detail below. For now, the important point is that the outer binary tree leaf nodes point to inner binary trees, and the inner binary tree leaf nodes point to files or chunks of RAM (“blocks”). So, by traversing the outer binary tree, we will find an inner binary tree, and then traversing the inner binary tree, we can find the correct file (which we will also call a block). Blocks may be split within conceptual hierarchies, so the information for Houston, Texas, might be in a separate block than the information for Dallas, Texas. This means that information for cells higher in the hierarchy may need to be stored in multiple places. So, the information for the Texas parent cell may be stored in two or more adjacent blocks, and the same goes for information for the United States parent cell. When this is a case, if we process an update for such a cell, we may need to create a new message so that it will be processed on the next block as well.

# Code Organization Overview

* The BlockIO folder contains an interface and classes for serializing information to blocks.
  + IBlockIO defines a simple interface for reading and writing blocks, and obtaining and releasing write locks on blocks. Each block is associated with a Guid (effectively the file name for the block).
  + SerializableBlockIO is a static generic class for serializing an instance of a class to a block or reading a block and producing an instance of a class. All serialization is done using the .Net implementation of protocol buffers.
  + InMemoryBlockIO is an in memory implementation of IBlockIO.
  + AzureBlockIO is an implementation of BlockIO that uses Azure blobs to read and write blocks.
* The CellAndBlockContents folder contains classes that define the contents of cells.
  + BlockContents contains CellInfos organized into a HierarchicalLocationTree (see below), as well as information about the next time at which an update must be made.
  + CellInfo defines the structure of cells.
    - As described further below, each CellInfo uses HierarchicalLocationTree both to store source dependency information and to list the targets of the cell (i.e., cells to which changes in the cell may need to be propagated)
    - Each CellInfo also stores a list of recent StreamingMessages, to prevent a single streaming message from being processed twice, as also described below.
  + CellValueBase is a base type for defining the value associated with a cell; this should help assist protocol buffers serialization, particularly for serializing generic types where the generic type must be serialized.
  + CellValueForBuiltInType inherits from CellValueBase and is a generic type consisting of a single generic item. Thus, we would use CellValueForBuiltInType<double> to represent a CellValue whose value is simply a double.
* The CellLocators folder contains classes for organizing and indexing cells (see also Cell Locators below for more information)
  + HierarchicalLocationList is a hierarchical list that can store objects at different parts of the hierarchy. Thus, HierarchicalLocationList<CellInfo> can be used to store objects at “United States”, “Texas”, and/or “Dallas”, and HierarchicalLocationList<bool> can be used to collect some number of hierarchical locations. Because the data structure is recursive, a HierarchicalLocationList object may represent an inner location list rather than the root location list, but it can access the parent and/or children HierarchicalLocationList.
  + LocationTree is the binary tree class, for both the outer and inner binary trees, used to organize all blocks
  + LocationTreeFile adds some additional information about the location tree to what can be serialized
  + ItemLocationRange defines the range of locations in a block. Note that the first and/or last location may refer to the middle of a hierarchy. But even if the first location corresponds to Dallas, Texas, the information about Texas and the United States would still be stored, with changes synchronized to both files.
* The CellMessages folder contains classes (described below in the “Cell Messages” section) that can be used to update source cells, propagate changes in cells internally to other cells, and export cell values.
* The Logging folder contains an interface and classes for a basic error logging functionality.
* The MessageDelivery folder contains
  + a low-level queue interface IMessageQueue, implemented by InMemoryMessageQueue and AzureMessage Queue, both of which will automatically return messages to the queue if enough time has elapsed without the message being confirmed as successfully processed;
  + a MessageQueueForInnerTree that uses the IMessageQueue to handle the queuing of AggregateMessages (which contain CellMessages)
    - This uses an AggregateMessageTracker class, which keeps track of which AggregateMessage contains each CellMessage and can create new AggregateMessages that are subsets of old AggregateMessages when some parts of an AggregateMessage succeed but others fail and need to be reissued.
    - It also uses AggregateMessageStatus class, which tracks the completion status of individual AggregateMessages
  + a MessageQueueForBlock. The InnerTreeManager (below) takes CellMessages from a MessageQueueForInnerTree and assigns them to a separate MessageQueueForBlock for each block represented by the inner tree. When it comes time to act on the messages it has queued, it calls the ApplyMessagesToBlock() method, which tries to read the block, call the ProcessMessages() method of the BlockContents, and then write the changed block.
* The CellCalculation folder contains
  + an interface ICellCalculator (to process particular types of cells), which includes a CalculateCell(HierarchicalLocationList<CellInfo>) method. Note that because the parameter is a HierarchicalLocationList, the CalculateCell method receives both the CellInfo and the specific spot within the hierarchical location list where this cell is located. The CalculateCell method will be called after messages, including custom messages, are processed.
  + various built-in implementations of that interface,
  + and a class CellTypeRegistration to register implementations defined externally to the class library so that novel types of cell manipulations (equivalent to a spreadsheet’s formulas) can be made.
* The TreeManagement folder includes an
  + OuterTreeManager (to handle reading and writing of the outer location tree),
  + InnerTreeManager (which calls the MessageQueueForInnerTree to load messages, distributes them to MessageQueueForBlocks, and then processes the queues that have the highest priority messages)

# Cell Messages

Updates of all types are made via instances of the CellMessage class. A CellMessage is used to set a cell to a value based on an external source, to update a cell based on a change made in one of its source dependencies, and to export a value to an external source. Inheriting from CellMessage will be the following (explained further immediately below).

* MessageToSetValueOfCell (which is used to update the output value of a cell at a particular location without changing dependencies; this applies only to source cells, i.e. those whose values are not dependent on any other cell and thus have no source dependencies);
  + MessageToSetValueAndOptionsOfCell (which derives from MessageToSetValueOfCell and also allows specifying cell options, such as whether children should automatically be recalculated when the parent’s value changes and whether this should be kept with siblings)
  + MessageToSetValueAndTargetsOfSourceCell (which derives from MessageToSetValueAndOptionsOfCell and also allows adding target dependencies)
  + and MessageToSetValueAndDependenciesOfCell (which inherits from MessageToSetValueAndTargetsOfSourceCell and may be used also to change a calculation type and alter source and target dependencies; this can thus be used for non-source cells, i.e. cells whose values are based on other cells);
* MessageToPropagateChangeInCellValue (which is used internally to send a message to a target dependency cell that a change has occurred in a source dependency cell);
* MessageToRequestCellValue (which is used by cell A to send a message to cell B requesting that cell B send a MessageToPropagateChangeInCellValue to A; this overrides MustRecalculateAfterMessageReceipt so that message calculation will not automatically occur)
* Custom messages
  + These can derive from CellMessage and can be used by a cell of a particular type to send information to another cell by some means other than sending its entire output value to the cell’s CalculationSources HierarchicalLocationTree. More information is provided below in Message processing and custom messages.
* and MessageToExportCellValue (which is used to send a message that an external data source must be updated based on a change in a cell).

Messages from one inner binary tree to another will be aggregated into an AggregateMessage, consisting of a List<CellMessage> property and an int totalMessageSize property.

Parent-child cell communication. If a parent cell is changed, then no message is required to change the value of child cells. Rather, the parent cell simply changes the MustCalculate property for the child cells, and they will be recalculated, thus looking at the value of the parent cell. However, if as a result of a child cell change, the parent or other ancestor must change, then a MessageToPropagateChangeInCellValue must be used. That is, the parent cell or other ancestor must be listed as one of the cell’s targets. If the parent cell is located in the same block as the child cell, this will be processed immediately, but if it is located in a previous block, then it will be added to the appropriate queue. Either way, we will ensure that if there are child cells in a later block that depend on this parent cell, the parent cell will be updated in those blocks (and can then set MustCalculate to true for their local children).

Export of data. Changes in some external database (a relational database or in a NoSQL data store) may mean that new data must be added to the block computation database or data must be changed in the block computation database. That is handled simply by adding MessageToSetValueOfCell, MessageToSetValueAndTargetsOfSourceCell or MessageToSetValueAndDependenciesOfCell messages to the appropriate queue. This will not be done by the block computation library itself, but might be executed by a web or worker role. Meanwhile, change in the data or calculation of new data in the block computation module may result in a need to copy information back to one or more other databases (such as some SQL or NoSQL database). The block computation module will simply add a MessageToExportCellValue message to one or more export queues. Some type of worker role (not defined by the block computation library) can read from these queues and process the export requests appropriately; we will not be implementing this. Note that other than this export capability (which requires only adding messages to queues), we are not currently planning to support a generalized query capability.

Cell initialize. When we initialize or reinitialize a cell, we must specify in a MessageToSetValueAndTargetsOfSourceCell or MessageToSetValueAndDependenciesOfCell object

* the cell output value (an object, which could be null, for example if the value is to be calculated based on some other cells),
* the cell location (LocationIdentifier[]),
* the calculation type to perform (a string, which should be “NA” if the cell’s value is not calculated but is instead externally set, and should generally be quite short to save space),
* any source or target dependencies that we wish to add or delete
  + Source dependencies to add are specified using HierarchicalLocationList<CellValueBase>.
  + Target dependencies to add and source and target dependencies to delete are specified using HierarchicalLocationList<bool> where the bool indicates that we have a dependency at the relevant level;
* a bool replaceAllExistingSourceDependencies (if true, then all source dependencies are deleted before the specific dependencies added are specified);
* a bool replaceAllExistingTargetDependencies; and a List<string> exportQueues. If B is dependent on A, then we say that A is the dependency source and B is the dependency target, so a change in A may result in a cascading change to B. When creating a cell, specifying the source and target dependencies is optional. When we create a cell with a target dependency, then the target will automatically be updated with the source dependency. That is, the cell that is created will send

If the MessageToSetValueAndTargetsOfSourceCell object refers to a location where the cell already is storing information about a source, then the new cell value will replace the cell value in that location. If replaceAllExistingSourceDependencies or replaceAllExistingTargetDependencies is true, then the corresponding dependencies will be replaced; otherwise, the existing source and target dependencies will be retained except for those specifically designated as to be removed. The new source and target dependencies will be added. When a cell’s source and target dependencies change, the class that performs the calculation on the cell may propagate messages to those dependencies causing them to make changes, possibly including changing the corresponding target or source dependencies.

Message propagation with MessageToPropagateChangeInCellValue. If B is dependent on A, then when the value of A changes, we must send a MessageToPropagateChangeInCellValue message to B so that B can incorporate that change. The message will include the location of A, the location of B, and the new value of A, along with the inherited priority value. (Note that A is an object and so it can contain multiple propertys, lists, vectors, etc., and so can B.) The message will be sent by being added to a queue for the inner binary tree containing B.

Large message handling. Note that the size of an AggregateMessage may be limited on Azure, so that implementation of the queue should check whether it needs to create a new AggregateMessage when adding a CellMessage. (Azure queue messages on the service bus brokered messaging queue have a 256K limit, but we should assume a 250K limit to account for header information.) The queue interface should also support a *single* CellMessage > 250K by creating a blob for the message (and referring to it with the type OversizedCellMessage); this blob would then be read and included in the list of cell messages when reading the queue message, and the blob would be erased before the queue message is erased.

# Cell Locators

This section describes both the HierarchicalLocalList and LocationTree data structures in more detail.

Location list: Within the file, we will organize various pieces of information (including lists of dependency sources and dependency targets) by location in a hierarchical list. So that we do not need to be repetitive, we will create a data structure HierarchicalLocationList<T>, structured as follows

* Dictionary< LocationIdentifier, HierarchicalLocationList<T> > Children; // the lists hierarchically below this one
* HierarchicalLocationList<T> Parent; // the parent of this list
* T ItemAtThisLocation; // the value at this list; may be a reference type or a value type, and can be set to default(T) (null for a reference type) if there is no item here.

So, for example, suppose one object is stored in “A A” and another in “A B” (where the A and B are the LocationIdentifiers). Then, this would look something like this:

* SubHierarchicalLocationLists
  + A
    - A
      * ItemAtThisLocation: Object in A A
    - B
      * ItemAtThisLocation: Object in A B
  + ItemAtThisLocation: Default(T) [e.g., null or false if T is bool], because there is no object at “A”.
* ItemAtThisLocation: Default(T) (because there will never be an object at the head of the list)

Note that if there are no items in a list, we will use null.

Ordinarily, a HierarchicalLocationList<CellInfo> can be split anywhere in the list, and splitting should generally try to make two items of equal size. However, the CellInfo class contains a KeepWithSiblings bool property. This should be made the same for all children of a parent. Where true, all the children must be kept within the same block. For example, if the Objects in A A and A B above were CellInfos with KeepWithSiblings == true, then the list could not be split between them. Also, if there were further SubHierarchicalLocationLists below A A or A B in the hierarchy (e.g., A B 1), we could not split between them either. This will be useful when calculation on one sibling depends in part on the value of other siblings. For example, if calculating the percentile of each of various items, the easiest approach may be to keep all the items in the same block. This approach must be used cautiously, however, where there can be a very large number of siblings.

Binary tree class: Our binary tree node class is LocationTree. Note that the outer LocationTree is used to locate inner LocationTrees, and the inner LocationTree is used to locate blocks, which themselves organize objects by location in HierarchicalLocationList<T>. The LocationTree should include a property representing a range of items to be included (or null if all items are included, for a root node), and recursive propertys to a left tree and a right tree. Ranges should be stored with the first value as included in the range, and the second value as excluded. So, for example, if the tree is A to M and the left node is A to G, then the second range value for the left tree (G) represents a value that would be included in the right tree, and the second range value for the tree as a whole (M) represents a value that would not be included in this branch of the tree. If the node is a leaf node, then the left and right trees will be null. For leaf nodes, there will be a Guid property pointing to the storage location (the block id, i.e. the filename or blob name) of the items at that level of binary tree. Also, each node will include a counter of the number of nodes beneath it in the tree.

Nesting of binary trees and queuing: As noted above, the outer binary tree will be used to store the locations of the inner binary trees. That is, a leaf node on the outer binary tree’s Guid property will point to a block containing an inner binary tree. The inner binary tree can then be traversed to find a location corresponding to a leaf node, and that leaf node will contain a pointer to a file containing the actual data. Corresponding to each leaf node of the *outer* binary tree, and thus to each inner binary tree as a whole, we will have a queue of items to be processed (e.g., items to be added to storage or changed in storage based on either external instructions or internal cascading changes). We will then read that queue into memory (or as much as can comfortably fit), group changes by the file to be affected, prioritize them, and load one file at a time to make changes. This minimizes the number of separate reads and writes needed.

Queue addresses and redirection. The name of each queue will be a hash of the location of the *first* item in the inner binary tree. The hash can be accomplished with GetHashOfSingleLocation in ItemLocationRange. As a result of this, if a node on the outer binary tree is split (producing two inner binary trees where there was previously one), a message sent by a process (such as a web server) that does not know of the split will continue to be delivered to a queue for an inner binary tree. That inner binary tree can then redirect the message to the next inner binary tree, after detecting that the message does not target a location in that inner binary tree. This is explained in more detail in “Inner tree management” below.

# Block structure

Structure of each block: The file is structured into a BlockContents data structure. This, along with each of the objects within it, must be serializable using Protocol Buffers (use the implementation at <https://github.com/mgravell/protobuf-net/>; see the post at <http://wallaceturner.com/serialization-with-protobuf-net> for basic instructions on how to use attributes to decorate classes, and the examples in the existing code and <http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual> for more detailed instructions). Note that an object can have related objects, which will be serialized if an appropriate attribute is set, but this is not required.

* ~~Shared source information. This is a store of information that is a source for more than one cell within the file. HierarchicalLocationList <object>. Note that a cell always has access to its parent and ancestor source items (through the Parent in the HierarchicalLocationList), so we do not need to use shared sources in that case. The locations are the~~ *~~source~~* ~~locations for each object.~~ Note: We are not going to implement this for now, but may implement it later.
* Cells HierarchicalLocationList <CellInfo>. The CellInfo class will include a
  + HierarchicalLocationList <CellValueBase> CalculationSources (for the source cells specific to that cell only)
  + ~~HierarchicalLocationList<bool> SharedSources (references to the shared source information). Note that the HierarchicalLocationList<bool> is used because there may be intermediate locations that do not have cells. In the illustration above, for example, there is no object at A (it is just the beginning of locations of other cells), so we would have false. So, if a shared source is A B, then we would have a single bool in the HierarchicalLocationList, thus indirectly referencing the shared source~~
  + List<StreamingMessage> StreamingMessagesAlreadyProcessed Internal messages that are to be stored temporarily with the cell, instead of simply having the contents of that message added to the contents of the cell (see below)
  + HierarchicalLocationList<bool> Targets (for the dependency targets of the cell)
  + HierarchicalLocationList<bool> TargetsToDelete (this indicates items that should be removed from the Targets message and receive messages indicating that this cell should be removed as a source dependency; for example, this might be appropriate in a calculation of PageRank if a web page was changed to delete a link that it formerly contained; this is not serialized, but is used internally)
  + List<string> exportQueues (listing any queues to which the changed data should be exported)
  + CellValueBase internalStorage (storing any information that this cell needs to make calculations; for example, we may store some aggregate numbers needed to calculate a standard deviation; or we may store the value the last time we propagated the cell, so that we propagate only when the value changes by at least some threshold)
  + bool allSourceDependenciesResolved (true if all sources needed for calculation are present and false otherwise)
  + bool mustCalculate (can be set to true by a parent to indicate that this must be recalculated based on a change in the parent)
  + bool mustPropagateToTargets (true if the value has been changed and thus must be propagated to its targets);
  + bool mustExport (true if the value has been changed and must be exported to the exportQueues; should remain false if no such queues exist)
  + bool mustDelete (indicates that this cell should be deleted)
  + bool mustDeleteAlongWithChildren (indicates that this cell and its children should be deleted)
  + DateTime? nextAutomaticUpdate (if not null, then the next time at which this must be automatically recalculated even if no input has changed);
  + and a CellValueBase object CellOutput, with the output (i.e., the value) of the cell. Note that this value can be any subclass of CellValueBase, including CellValueForBuiltInType or a class with its own properties (and optionally methods).
* DateTime? NextAutomaticUpdate. The soonest next automatic update (see below for an explanation) later than the current time, or null if none.

# Inner tree management

InnerTreeManager. A class InnerTreeManager will continuously handle messages for the inner binary tree by reading this queue. It reads up to maxMessages (say, 2000) simultaneously from the queue. It should then re-read the queue every readFrequency = 1 seconds (even while files are being processed), but no more than the number of messages needed to get back up to maxMessages (and it should skip a read if the number to be processed is equal to maxmessages). InnerTreeManager associates messages to be processed with a local in-memory queue for the corresponding leaf node of the inner binary tree (i.e., each separate file). If the InnerTreeManager receives any messages that should have been provided to a different inner binary tree, then it creates new messages for the queue corresponding to that inner binary tree, and it marks the messages that it processed as successfully processed.

Determination of which file to process. InnerTreeManager will continuously process files with pending queue messages. All I/O processing must be asynchronous using C# async functionality. When adding a message to a local in-memory queue, it adds the message’s priority to the sum of priorities for the file.

Simultaneous processing of files. Currently, InnerTreeManager processes up to maxSimultaneousFiles (say, 5), and this is fixed. We should change it so that InnerTreeManager processes as many files as possible consistent with memory availability, and so that maxSimultaneousFiles is variable. Initially, only one block should be processed at once (maxSimultaneousFiles == 1), and we should consider increasing maxSimultaneousFiles only after complete processing of a block. We should also include variables for maximumMemoryUsagePerBlock and averageMemoryUsagePerBlock. The InnerTreeManager will periodically update these, subtracting used memory before any blocks were processed, and dividing by the number of current blocks to determine current memory usage per block, then updating maxima and average. When determining whether an additional block can be processed, we should look at the available memory and ensure that it is greater than maximumMemoryUsagePerBlock \* memoryCushionMultiplier. We can experiment with different memoryCushionMultipliers to see what will generally be sufficient. We may accomplish this with .Net’s MemoryFailPoint class. We should also gracefully handle OutOfMemory errors by abandoning processing of particular blocks. We should ensure that should these occur, memory associated with the block processing is released, but the in-memory queues are unaffected.

Automatic updates. Usually, a cell is updated only as a result in the change of the value of its source dependencies. However, some cell’s values may be dependent on time. For example, a cell might be used to aggregate the number of sales of a good over the past hour. This may be updated when there is a new sale but also must be updated when an old sale is no longer within the past hour. As a result, each cell can have an AutomaticUpdateTime associated with it, and the corresponding block will then have the earliest of these times as its own automatic update time stored as NextAutomaticUpdate. InnerTreeManager keeps track of the NextAutomaticUpdate for each file; when InnerTreeManager first loads, it should assume a NextAutomaticUpdate of DateTime.Now for each file, since each file may require an automatic update. A non-null NextAutomaticUpdate that is before DateTime.Now initially counts for a priority of 1.0. However, each time the InnerTreeManager begins to process a file it should multiply the priority attributable to NextAutomaticUpdates for all other files by nextAutomaticUpdateMultiplier = 1.1, so that as time moves along, accomplishing a timed update will become higher priority.

Requeuing continued messages. A message may be processed for a location that spans blocks because there are hierarchically lower locations than the target location. For example, data about Table 1 might be included in various blocks, because there are cells for Row 1 to Row 100,000,000 of Table 1, and that may be too many rows to fit in a single block. We need to maintain the data about Table 1 itself in each of the blocks, so that each row can access information about the table in making its calculations. (Recall that information from hierarchically superior items should be available during calculations.) We accomplish this with the InnerTreeManager’s ConsiderPropagatingMessageToAdditionalBlocks method. This looks to see whether there are other blocks within the same inner tree that include locations that are descendants of the targeted location. If so, the message queue for each of those blocks is stored in a non-serialized AdditionalTargetBlocks property of the CellMessage. When the AggregateMessageTree is told that a message is complete, it will send the CellMessage targeting the same location to the *next* target block. Meanwhile, if the next inner tree includes blocks that must receive the message, then a new AggregateMessage is added to the next inner tree’s queue.

Why does ConsiderPropagatingMessageToAdditionalBlocks update the AdditionalTargetBlocks property instead of simply doing the queuing directly? The concern is that we need to know when we have completely processed an external queue message. So, if we have a single Azure queue for an inner tree, and we receive a message that needs to be handled on blocks 1, 2, and 3 of that inner tree, then we want to tell the Azure queue mechanism that the message is complete only after we've done all three blocks. Of course, we're using only our in-memory queues to allocate the message to blocks 1, 2, and 3. If we add to our in-memory queues for block 2 and block 3 right away, then 1, 2, and 3 might complete in any order, and we would need a mechanism to determine whether all three blocks have completed processing before telling Azure that we have completed the message. Note that if we realize that we need to send the message to another inner tree altogether, then we add a new message to the Azure queue for the next inner tree. We can't with our current addressing system use the Azure queue to enqueue a new message for block 2, because the address initially takes us to block 1.

An alternative might be to make something like the following changes: Store in CellMessage the number of inner tree blocks that a message must be processed on. In AggregateMessageStatus and AggregateMessageTracker, change the MarkCellMessageAsComplete so it just decrements the number of remaining inner tree blocks to process when a cell message is marked as complete. Then, the ConsiderPropagatingMessageToAdditionalBlocks method could, instead of creating an AdditionalTargetBlocks collection, change the CellMessage to indicate the number of blocks that must be processed and then enqueue them directly. This adds some complexity, and you would need to add tests to make sure that we are doing the aggregate message tracking correctly. But you can work out this approach if you think it would be better. Certainly an advantage is that the message might end up being fully processed more quickly.

# Block processing

Processing of block. When the InnerTreeManager begins to process a file, it moves the messages for that file to be processed to a separate local in-memory list of messages being actively processed. That way, new messages that come in while the file is being processed can be queued for the next time the file is modified. It will call the MessageQueueForBlock’s ApplyMessagesToBlock method, which (as noted above) downloads the block and deserialize its contents using the SerializableBlockIO<BlockContents> class into a BlockContents Object and then process all the messages from the actively processed queue by calling the BlockContents’s ProcessMessages method. ProcessMessages begins by processing each message (as described below).

Propagation. After it does this, it creates the new messages that must be propagated. First, it looks at the CellInfo’s TargetsToDelete, and removes these from Targets, creating messages to the former targets indicating that they should remove this CellInfo as a source. Then, the processor must look for mustPropagateToTargets = true, create messages processing the cell value, and adding them to the list of messages. Then, it separates all separates all the messages into messages within the file and messages external to the file. It then processes messages within the file, continuing to modify the file contents and look for new messages until there are no more local messages. (If, after doing this maxTimeToApplyInternalMessages = 5 times, new local messages are being produced, then these local messages should be bundled with the external messages, so that the loop does not continue forever or excessively long.) It then sends the external messages, grouping them into different queues (based on the routing information in the outer binary tree). It also sends all export messages (by checking and changing mustExport), again grouping by queues. If these approaches are successful, it saves the changes to the objects, and then after that it will call a method to dequeue all messages from the local in-memory queue. This must also result in the messages being marked as successfully completed in the broader queue corresponding to the inner binary tree.

Handling of exceptions for a block. If an exception occurs in the handling of a block (for example, as a result of an intermittent problem with the queue service), then the local in-memory queue of messages to be processed and the local in-memory queue of messages being processed should be cleared. However, the messages should not be marked as processed successfully from the source queue for the inner binary tree, because they need to be processed again. Instead, the queue service should be informed that processing failed, so that the messages can be requeued. For each block, we will keep track of an EarliestRetryTime. We can set this initially (after the first exception) to DateTime.Now plus CurrentRetryInterval = InitialRetryInterval = ten seconds, but we should implement exponential backoff so that CurrentRetryInterval increases each time by a multiple of 1.5. (Once we successfully process a file, we reset CurrentRetryInterval = InitialRetryInterval.) When DateTime.Now < EarliestRetryTime, we should not put any of the pending queue messages for that file into the local in-memory queues. We should log the exception when it occurs.

# Streaming messages

Aggregation of data without permanent storage. Sometimes the goal will be for target cell B simply to aggregate A without storing it. For example, if B is a sum of many cells, it may not make sense to recalculate the sum every time there is a new cell to include in the sum. This, however, leads to the problem of ensuring that the same message is not processed twice. A message can be processed twice for three reasons. First, A may send a message (e.g., using Azure’s queue service) but then the update to the data containing A may fail, so A will try to send a message again later. Second, B may receive a message and process it, but the attempt to confirm that the message has been processed with the Azure queue service may fail, and so the message will later be received again. Third, it may take a while for a message to be processed, and the queue mechanism may have thus restored the message to the queue in the interim. This can occur because we process files in our inner binary tree based on priority. Time is a factor in this priority calculus, and we may adjust the time before a message is returned to minimize the risk of processing messages that have already been returned to the queue again.

For the typical case, we do not need to worry about double processing, because the calculation is idempotent; that is, there is no harm from redoing the calculation. However, if the dependency target is aggregating the dependency source without permanently storing it, then we must ensure against double processing by retaining the message. The message should be retained in StreamingMessages until the following happens at least minBatchesWithoutSameMessageBeforeDeletionOK (= 100) times: a batch of messages being processed by B includes a message from the same inner node as the original message BUT does not contain the same message again. The StreamingMessage class includes the CellMessage, a bool indicating that the message has been processed, and a count of the number of batches remaining; when this number gets to zero, it will be deleted.

The StreamingMessage class includes a bool MessageHasBeenProcessed. Thus, when the cell’s CalculateCell method is called, it can ignore StreamingMessages that have already been processed and consider only those that have not been processed. It can obtain all of these messages by calling CellInfo’s GetUnprocessedStreamingMessages method.

# Message processing and custom messages

Overview of processing. Once a block is being processed, then each message concerning that file must be processed as well. First, all messages for a particular cell are processed, and then the cell is calculated. The processing of messages is accomplished by descendants of the CellMessage class, and the cell calculation is accomplished by a cell calculation interface. Typically, the new value of A will be stored along with B, and B will then be recalculated based on that value and values of other source items stored with it. After the calculation, the message will be automatically propagated to its targets if MustPropagateToTargets is set; ordinarily, this is set when the message is calculated, but if a message directly changes the cell’s output value, then it can set MustPropagateToTargets directly.

Built-in message types: For the built-in message types, all of the work will be handled by the database. For example, if an updated source dependency value is specified, it will automatically be copied to the CellInfo of the target.

Custom message types

* Explanation of use. However, often it will be desirable to have custom message types. For example, it might not make sense for a source dependency that has a large value object to copy that entire object to the target dependency every time it is modified. Or, it might make sense for some information from the source object to be stored in the InternalStorage of the CellInfo, instead of in the CalculationSources. This might be the case if the information is easier to access and process through the InternalStorage of the CellInfo, rather than having to navigate the HierarchicalLocationList.
* Implementation.
  + The custom message type must define ApplyMessageToCellInfo, so that when it is received, the target CellInfo can be updated appropriately. ApplyMessageToCellInfo might simply modify the CalculationSources property.
  + The custom message type also should override MustRecalculateAfterMessageReceipt, to indicate whether the cell must be recalculated after the message is received.
  + The custom message can inherit from MessageToPropagateChangeInCellValue or from some other message type, if it is desirable for the ApplyMessageToCellInfo to call the same method on the base type. This will often not be the case, however, since the base method will copy the output value of the source to the target.
* Definitions. ApplyMessageToCellInfo and MustRecalculateAfterMessageReceipt.
  + public abstract void ApplyMessageToCellInfo(CellInfo targetCell);
  + public virtual bool MustRecalculateAfterMessageReceipt()
* Example. Suppose for example, that we are storing states with each of their cities and the city populations. It may be desirable to store with the state information about the largest city. This might be useful, for example, if each city contains a field populationRelativeToLargestCityEver.
  + It would be cumbersome to have to search through all of the cities to find the largest each time this information was needed, and also some cities may be stored in a separate block.
  + Thus, when a city has its population increased, it can check the parent object to see if its population is greater than the largest ever city in the parent. This check would occur in the CalculateCell method for the message processor for the city cell, and if a message needed to be sent, CalculateCell would return it.
  + The reason to send this as a message, rather than simply modifying the parent directly, is because the first instance of the parent might be located in another block. In that case, the message will be routed automatically through an external queue.
  + The message that is sent by the child to the parent would be a custom message, perhaps of type PotentialLargestCityMessage, and it would update the parent message’s CellInfo in this way. In this example, this would be preferable to a StreamingMessage, because a StreamingMessage would be stored in the cell for a period of time. That is not necessary here, because operation is idempotent.
  + When the parent gets the message, it can see whether this is in fact a new largest city. (It might not be if the parent was updated in the interim.) If so, the parent can update its value. As long as the state’s AutomaticallyRecalculateChildrenWhenChanged is set to true (this should be set when the state cell is originally created by using MessageToSetValueAndOptinosOfCell), then its city children will be set to recalculate by ProcessMessages after the state is calculated. Note that if the children are located in multiple blocks, the message will be propagated to the parent at all of these blocks, so all of the children can be updated.
* Example with Map/Reduce.
  + Suppose that we did not want to store the largest city ever, but the *currently* largest city. This is more complicated because if the leading city goes down in size, we must look at all of the other cities to see if any of them is now the largest. Because the cities might be on multiple blocks, we would need to look on each of these blocks to find the largest city and then compare all of these results. In effect, we would need to implement a simple Map/Reduce algorithm. One way this might be done would be to send the parent a custom message of type QueryForLargestCity. Each instance of the parent would then send a custom message of type PartialResponseToQueryForLargestCity with the results to some location (possibly itself). It would then store this partial response in the InternalStorage. Once a parent node received responses from each instance of the parent, it would then be able to calculate the result and delete the intermediate data from InternalStorage.
  + To support this, we should design:
    - a helper class BlockRangeCollection whose methods can help determine whether all of the partial results have been received. Each responding node can send information about the range of the block within which it is contained, and this information can be added to a List in the BlockRangeCollection. Then a method RangeIsComprehensive could determine whether these responses are comprehensive. (Must be implemented)
    - Alternative mechanisms for propagating messages beyond the first block to receive the message (implemented). Currently, AssignMessageToBlockManager calls ConsiderPropagatingMessageToAdditionalBlocks. We could create an enum on CellMessage AdditionalBlocksPropagationBehavior that would determine the appropriate behavior: normal (additional blocks are added to AdditionalTargetBlocks to be sent after processing is complete along with other messages, or the message is sent to the next inner tree if that is where the next target is), doNotPropagate (which would allow the first recipient to process the message all by itself, without storing the message in other blocks), skippingFirstBlock (which would result in the message being skipped by the first block and processed only by subsequent blocks).
      * For example, the PartialResponseToQueryForLargestCity messages might be sent with value doNotPropagate. Then, once the instance that receives the messages has all of them and processes them, it will send a message with information about the final calculation to itself with skippingFirstBlock. (AddGeneratedMessageToAppropriateList would detect this in a new message generated and ensure that the message was added as an external message. We would then need our code )
      * a mechanism for sending a message only to the *first* instance of a node, and then another mechanism to send a message to only the remaining instances of a node. That way, the first node could do all of the aggregation of the query responses. Then, when it has a result, it could send a message with the query result to all of the other nodes.
  + More detail for map-reduce example
    - The process would begin when a city cell realizes (i.e., the CityCellProcessor's CalculateCell method recognizes) that the change in its value may necessitate recalculating the state cell. This might be because the old value was the largest city value for the state (whose value can be accessed directly) and the new value is lower, or because the new value is greater than the largest city value for the state. The CalculateCell method would then send either an IncreaseLargestCityMessage (if the value has gone up) or a QueryForLargestCityMessage (if the previous top value has gone down) to the state cell.
    - The IncreaseLargestCityMessage is easy to handle; the message processor would check whether the new value was greater than the state cell's value. If so, it would increase the state value, set MustPropagateToTargets to true, and that's it.
    - It's the QueryForLargestCityMessage that initiates the query that we are interested in. The message's ApplyMessageToCellInfo would modify CellInfo's InternalStorage to indicate that the largest city needed to be updated.
    - Then, the StateCellProcessor’s CalculateCell method would see that flag, reset it, and then examine all of its children to determine the largest.
      * If locationIsFullyContainedInBlock, then there is no need to send any more messages. It would just set CellOutputValue (and then we could skip all the remaining steps).
      * We thus assume below that the location is *not* fully contained in the block.
    - Then, the CalculateCell would return a partial response message PartialResponseToQueryForLargestCity.
    - This partial response should be set to deliver only to first instance of state cell (i.e., AdditionalBlocksPropagationBehavior would be set to doNotPropagate), and its ApplyMessageToCellInfo() saves partial value into a BlockRangeCollection stored in InternalStorage. Because it does not override MustRecalculateAfterMessageReceipt, MustCalculate will be set to true.
    - Thus, the StateCellProcessor's CalculateCell method will run, and it uses BlockRangeCollection to find out when all responses are received, and once they are, gets the largest from partial values.
    - It then clears the BlockRangeCollection and saves the largest value as the CellOutputValue. As long as AutomaticallyRecalculateChildrenWhenChanged was set for the state cell when it was created, this will automatically result in the MustCalculate flag being set for each of its child cells. The city CalculateCell methods will thus be called, and the population ratios can be recalculated. (This illustrates an advantage of the block computation database; no messaging is needed for this step.)
    - But this message also must be propagated to the other instances of the state cell. This will not happen automatically, because the cell is not a target of itself (a situation which would create an infinite loop). So, we create a MessageToSetValueOfCell message with AdditionalBlocksPropagationBehavior set to skip the first cell. When the ApplyMessageToCellInfo method for this message is called, it should simply change the cell value. Again, the children will be set automatically to recalculate.

# Cell calculation

We will define in an interface ICellCalculator a method to perform the final calculation on the cell. (Note that it might seem that we should have ICellCalculator be an interface for a static method, but c# doesn’t allow that.) The method is:

public List<CellMessage> CalculateCell(HierarchicalLocationList<CellInfo> theCellLocation, bool locationIsFullyContainedInBlock)

* The CalculateCell method is responsible for setting the allSourceDependenciesResolved property, the mustPropagateToTargets property, the mustExport property, and the output property of CellInfo.
* If locationIsFullyContainedInBlock is true, then the cell *and all of its children and descendants* are located in the cell.
* If CalculateCell changes CellOutputValue and the cell’s AutomaticallyRecalculateChildrenWhenChanged flag is set, then the children’s MustCalculate flags will automatically be set to true.
* CalculateCell should return a list of messages to send, including custom messages and StreamingMessages, if any. If there are no such messages, it should return null (or an empty list). Note that if the value of the cell is to be copied to the target’s CalculationSources, that does not require a custom message, so null can be returned.

Note that the objects in all sources should contain all information needed to identify how those objects are relevant; for example, an enum property may be used to identify the type of source item, or the type of the item may contain the relevant information. The CalculateCell method is not passed the *location* of any source object, since that is usually not relevant. Objects from hierarchically superior items that are relevant to calculation can be looked up directly from the HierarchicalLocationList.

Registration of ICellCalculators. The ProcessMessages method must be able to determine which instance of the ICellCalculator to use to process a particular cell. As a result, we will create a class CellTypeRegistration, with a method RegisterICellCalculator(string calculationType, ICellCalculator instance), and a method ICellCalculator GetICellCalculator(string type), which returns null if none has been registered. The CellTypeRegistration class internally will use a Dictionary<string, ICellCalculator> to store the ICellCalculators. The CellTypeRegistration class can be passed from the worker role to the InnerTreeManager and the ProcessMessages method of BlockContents. This overall design will allow the block computation module to be stored in a class library, and other code that uses the calculation module for specific types of calculations to define additional types of calculations.

Built-in calculation types. We should define ICellCalculators for various common calculation types: Count, Sum, Minimum, Maximum, Average, Standard Deviation, Weighted Average, and Weighted Standard Deviation. All of these must be defined separately both for use with permanent storage (i.e., where each item being aggregated is permanently stored in the block) and without. The versions not using permanent storage will store internally aggregates that are updated with each new number (I have a StatCollector class that can accomplish this). Each should use brief (no more than three letter) CalculationType names. In the case of Sum, Min, Max, Average, and Standard Deviation, the source cells can be objects of type int, float, or double, and the output type will be float or double if any of the inputs are. For the weighted average and weighted standard deviation, there must be an input type NumberWithWeight (with Number and Weight double propertys). It’s important to see how multiple cells are needed to make this sort of calculation. There might be one cell with the number and another with the weight, specifying yet another cell with the NumberWithWeight type as its output type as a target, and the NumberWithWeight cell in turn will in turn specify the calculation aggregator as its target.

Distribution built-in calculation type: We will also define a calculation type for approximating a distribution without permanently storing all of the items in the series for which we are storing a distribution. One input would be the number of items to store in the distribution. Suppose this is 1000. Then, when fewer than 1000 items are in the distribution, if a new item is added, then it is simply added to the array of items in the distribution, sorted appropriately. When the 1001st item arrives, it has a 1000/1001 probability of being added to the distribution, and if it is added, then another item chosen at random is removed. More generally, when item n arrives where n > numItems, then there is a probability numItems / n that this will be added. Also, the distribution cell should be able to receive a message indicating that a value has changed. For each item stored in the distribution list, we also store internally a Guid, so if a message indicates that a value has changed, we can look at whether that item was already in the distribution, and if so replace it. Finally, we should support removal of an item. If the item was not actually used to make up the distribution (i.e., the Guid is not included), then the removal message will be ignored; otherwise, the distribution will involve one less element.

# Block and tree node splitting

Splitting of binary trees: We will split a leaf node in an inner binary tree (i.e., add two children to the node, and move the contents to those two children, deleting the original file and its ID) when any block exceeds the maximum size (maxBytesPerFile). Splitting need not be instantaneous, however, so a block can exceed the maximum for some period of time. The maximum is simply that largest amount of memory that we want to process at a time. 64MB may be a good value, since one can read that much as a single transaction in Azure, or perhaps something slightly smaller, such as 60MB, would make sense to allow for the possibility of files that are temporarily larger. The split will occur at the node on the inner binary tree whose file is too large. We will call this an *inner split*.

After we split the inner binary tree node, thus increasing by 1 the number of leaf nodes in the inner binary tree, we will look at the number of leaf nodes in the inner binary tree. If this exceeds a constant (maxFilesPerQueue), then we will move the highest split in the inner binary tree to the outer binary tree. This is called an *outer split*. For example, suppose that a leaf node of the outer binary tree is 1,000,000 to 2,000,000. (I will use numbers rather than other types of LocationIdentifiers for simplicity of exposition.) Then, the inner binary tree head node might contain a split assigning the 1,000,000 to 1,300,000 range to the left and the 1,300,001 to 2,000,000 to the right. There may be many splits below that, but once there are too many leaf nodes in the inner binary tree, then we must convert the inner binary tree into two entirely separate inner binary trees (that is, we must delete the original inner binary tree file and create two new binary tree files), and we will split the corresponding node in the outer binary tree so that it is no longer a leaf node corresponding to the original inner binary tree, but instead a node pointing to two leaf nodes, one with 1,000,000 to 1,300,000 on the left and one with 1,300,001 to 2,000,000 on the right, each corresponding to an inner binary tree. Note that if some process sends a CellMessage to the original inner binary tree, it will now be processed by the 1,000,000 to 1,300,000 binary tree and redirected if necessary to that node’s sibling (with continued redirections if necessary as a result of rapid splits).

DeleteFile notations. When we split a binary tree, we will often need to create files (representing the new trees). We must make sure not to be left with orphan files if there is some failure before the creation process is absolutely complete. Before creating any file, we will put the name of that file into a list (at the end of the binary tree file) of files to be deleted. If we ever detect such notations, that will indicate that these files are not needed (perhaps because there was a failure along the way) and should be deleted if they exist, followed by a change in the inner binary tree file if possible to reflect the deletion. To conclude a change process, three things must simultaneously happen (that is, in a single atomic write to data storage): we change the binary tree so that it refers to the new files, the files are removed from the list of files to be deleted, and we add to the list of files to be deleted any files that will no longer be relevant (i.e., the file corresponding to the old node). Those files will then be deleted at a later time.

Process to accomplish an inner split: We will attempt to do all of these in order, obtaining a write lock on the inner binary tree file. (This is not strictly necessary if our assignment process works correctly, but it provides some additional protection. The write lock should last only during the splitting process, since that is the only time the inner tree file is changed.) If something fails, this process is idempotent and thus should work the next time around.

* Process all messages in memory for the inner split. Note that we will not request to receive any more messages from the queue while going through this step.
* Figure out the split location and the two Guids for the new block contents. A block should be split at the highest level possible. So, find the first place in the hierarchy where there is more than one item in the block. For example, if every node in the location tree starts with {A,B,C}, then you would split at the next highest level half way through. For example, if the next level included {A,B,C,D1-D100} We would split at {A,B,C,D500}. Note that this does not necessarily produce even split. If we had {A,B1} with no children and {A,B2} with hundreds of children, we would still split {A,B1} into one block and {A,B2} into another. This has the advantage of tending to keep related information together. Of course, if {A,B2} is large, it may soon need to be split again.
* Add the DeleteFile notations for these new blocks to the inner binary tree file.
* Write the two block contents files.
* Perform the last step indicated above (modifying the binary tree to refer to the new files, removing the new blocks from the list of files to be deleted, and adding the old block to the files to be deleted).
* Report that the command to conduct the split has successfully been processed.

Performing an outer split. An outer split again is the splitting of a node on the outer binary tree file (i.e., splitting the inner binary tree itself into two inner binary trees). This may be done in a similar way, though, as explained further below, only a worker that is the assignment manager can conduct an outer split, because two files are affected. If a worker role that is the assignment manager finds that a node on the outer binary tree should be split, then it should do the split. The assignment manager actually accomplishes the split by adding files to be deleted references to the outer binary tree file before creating the two new inner binary tree files. After the two new inner binary tree files are created, we can rewrite the outer binary tree file, including a reference to delete the old inner binary tree file. When this is processed, the old inner binary tree file is deleted.

# Worker task assignment

Coordinating inner binary tree instances. The algorithm for querying for data for each inner binary tree and handling that data is described above. We also need to be able to start each inner binary tree process and to share the processing of inner binary trees across different worker role instances, so that we can scale up easily. We will do this by storing information in the file containing the outer binary tree with information on which instances are responsible for each node. These changes will occur only when instances start up or shut down or otherwise fail to load.

Assignment of outer binary tree nodes (i.e., inner trees) to instances. Each worker (e.g., Azure worker role) needs to be able to know which nodes of the outer binary tree (i.e., which inner trees) it should work on. Our solution to this will be to have one worker serve as AssignmentManager function. When a worker starts, it should attempt to obtain a write lock on the outer binary tree file. If it can obtain the write lock, then it is the assignment manager. Otherwise, it will send a message to the assignment manager queue (which is independent of all the other queues described here) announcing its presence and the location of a queue for its assignments. The assignment manager will then send messages to workers dividing the outer binary tree nodes (i.e., the inner binary trees) among them. Each message will provide a list of nodes to work on, along with a DateTime? dontStartUntil time and DateTime mustFinishBy time, which should be separated by assignmentLength = 30 seconds. When the assignment manager first starts, it should set dontStartUntil = assignmentLength + TestableDateTime.Now, since workers might still be working on previous assignments, and we don’t want two workers receiving assignments at the same time. The message from the assignment manager should also include a DateTime lastOuterBinaryTreeChange, so that the workers can recognize that they need to update their copy of the outer binary tree if it has changed (so that they can route outgoing queue messages appropriately).

Each worker should send periodic check-in messages to the assignment manager queue, and these messages should include a pendingQueueMessageCount property, so that the assignment manager knows how busy the queue is. The worker should send these checkins every checkinLength = 10 seconds, and the assignment manager will respond to each checkin by sending a new work assignment queue message to the worker (which may be the assignment manager itself, as the assignment manager will also do the usual work). Ordinarily, this work assignment will be the exact same as the previous work assignment but with an extended time period, with dontStartUntil == null (reflecting that the worker has already started and thus need not wait to start) and mustFinishBy = TestableDateTime.Now + assignmentLength.

But sometimes, the assignment manager will reassign a node. It may do this because a worker role hasn’t checked in and its last mustFinishBy time has elapsed. In that case, the worker is assumed to have stopped. Or, if one worker has significantly more work than another, then some of its nodes may be reassigned. Or, as noted above and discussed further below, node reassignment should occur if the assignment manager receives a notification that an inner tree is to be split (requiring a change to the outer tree node referring to this inner tree). (No node reassignment occurs if a node on an inner tree needs to be split.) When a node is reassigned, the assignment manager will omit the node from the list of nodes that the worker who formerly worked on that node received, and will add it to the list of nodes that the worker who gets that node should receive, but with a dontStartUntil time that is after the mustFinishBy time of the former worker, so we don’t have any overlap.

When a worker receives its work assignment, it should update its own internal list of nodes to work on, specifically the mustFinishBy times. Meanwhile, if any mustFinishBy time is within timeToAbortProcess = 5 seconds, then it will stop working on that node, no longer keeping it in the rotation of nodes to tell to do a little work. The length of timeToAbortProcess should be set so that it is high enough so that any current processing on a block can be completed with enough time for the abort to occur simply. All of the queue items remaining once it is complete should be marked as failed, so that the items are immediately returned to the queue. (If this does not occur, then the queue mechanism will eventually return the items to the queue.)

Splitting outer binary tree. If the outer binary tree needs to be split as the result of a change in an inner binary tree (i.e., we need to split the inner binary tree into two inner binary trees), then the worker on the inner binary tree sends a message to this effect to the assignment manager, if the worker is not already the assignment manager. The assignment manager should wait for the mustFinishBy time to elapse, and then it can accomplish the outer binary tree split (necessitating changes to both the inner tree files and the outer tree file) itself. This approach should eliminate the risk of contention on the inner and outer tree files; while the assignment manager is splitting the trees, it will not do any other writing to the outer tree file, and no one will be writing anything to the inner tree file. Once the outer split is complete, the assignment manager will assign the two inner trees according to the usual assignment process. Note that as described above, it is important for the process to be designed so that if there is a failure at some step in the process, it will still be seamlessly completed.

Global queue. The assignment manager will also manage a single global queue. This can be used by processes that do not have access to the outer binary tree but need to send messages to particular addresses. The assignment manager will process messages in this queue by routing them to the appropriate inner binary tree. However, when a worker process processes a message, it will have access to the outer binary tree file (rereading it whenever it changes), so it can always direct a message directly to the appropriate external queue for the inner tree.

# Testing and interfaces

Interfaces and dependency injection for external services. To improve testability, we should use interfaces and dependency injection for (1) file storage (IBlockIO); (2) queues (IMessageQueueForInnerTree); and (3) logging (ILogger). Classes implementing the file storage interfaces should support Azure blob storage (using an Azure block blob, which allows a blob of up to 64MB to be written in a single operation); and an in-memory approach (which would simply allocate arrays of bytes). The queue interfaces should be implemented by a class using an Azure service bus brokered messaging queue (basic tier) (see http://msdn.microsoft.com/en-us/library/azure/hh367512.aspx), as well as an in-memory version. The queue routine should allow queuing of a List<CellMessage>, since we will be aggregating all messages from inner binary tree X to inner binary tree Y into a single queue message. Finally, classes implementing logging should use Azure table storage and an in memory implementation. For all of these, all calls should be asynchronous.

Unit testing. We should create appropriate unit tests to ensure that the code works properly. The unit tests can be tested easily by using the in memory versions of various classes, but should also be testable on Azure.

Integration testing. We also should create some robust integration tests that are designed to test the maximum capability of the system. That is, we should design a somewhat complicated spreadsheet using many target dependencies and test how many changes we can process at once. We will design a further test of more direct relevance to the project soon.

# Steps to take

Some of the basics of the code are already complete, though most of the code is untested and therefore may need to change). Here is a preliminary list of steps that need to be taken:

HierarchicalLocationList: Add tests to make sure that all methods work properly.

BlockIO: Add some simple tests of serialization using InMemoryBlockIO, making sure that protocol buffers are working correctly.

MessageDelivery: Add tests for all files (using the InMemoryMessageQueue) to ensure that the queuing and message routing works correctly and that AggregateMessages are handled properly. Add code to implement RouteExternalMessages, and tests.

InnerTreeManager. Add tests.

OuterTreeManager. Write code for the outer tree management, adding additional files and classes as needed for the assignment manager functionality. Add tests.

CellMessages. Add tests for ApplyMessageToCellInfo for each CellMessage subtype. Implement OversizedCellMessage.

BlockContents. Add code to implement MustPropagateToTargets and MustExport and TargetsToDelete. Then add tests for ProcessMessages.

CellCalculation. Test whether CountSources and CountStream work properly (possibly in conjunction with BlockContents above). Test other existing classes. Add code for the other cell types (average, min, etc.) discussed above; this code should follow the approach of CountSources and CountStream fairly closely.

Integration test. Create a test of the functionality with some spreadsheet design that uses all of the relevant features. For example, create large numbers of source cells at different points, and calculate statistics based on the numbers in each group, have these statistics target other cells so that we can calculate statistics on different sets of statistics (e.g., standard deviation across groups), export some final statistics. Later, we will create a test that is more specific to the project that we are working on.

Azure. Add Azure implementations of all the functionality currently implemented in memory, and add to the worker role.