Item 202
git_comments:

1. * * Offset of the field in the bucket header indicating the bucket's partition.
2. * * The number of buckets in the current table. The bucket array is not necessarily fully * used, when not all buckets that would fit into the last segment are actually used.
3. * * @return * @throws IOException
4. * * The recursion depth of the partition that is currently processed. The initial table * has a recursion depth of 0. Partitions spilled from a table that is built for a partition * with recursion depth n have a recursion depth of n+1.
5. // check that the memory segment book-keeping did not go wrong if (DEBUG_CHECKS) { HashSet<MemorySegment> segSet = new HashSet<MemorySegment>(); for (int i = 0; i < this.availableMemory.size(); i++) { MemorySegment seg = this.availableMemory.get(i); if (seg == null) { throw new RuntimeException("Bookkeeping error: null booked as Memory Segment."); } if (segSet.contains(seg)) { throw new RuntimeException("Bookkeeping error: Available Memory Segment booked twice."); } segSet.add(seg); } HashSet<MemorySegment> wbSet = new HashSet<MemorySegment>(); Iterator<MemorySegment> wbIter = this.writeBehindBuffers.iterator(); while (wbIter.hasNext()) { MemorySegment seg = wbIter.next(); if (seg == null) { throw new RuntimeException("Bookkeeping error: null booked as Memory Segment."); } if (segSet.contains(seg)) { throw new RuntimeException("Bookkeeping error: Write-behind buffer also occurred as available memory."); } if (wbSet.contains(seg)) { throw new RuntimeException("Bookkeeping error: Write-behind buffer booked twice"); } wbSet.add(seg); } }
6. * * The number of bytes for the serialized record length in the partition buffers.
7. * * Offset of the field in the bucket header indicating the bucket's status (spilled or in-memory).
8. * * Constant for the bucket status, indicating that the bucket is in memory.
9. * * Offset of the field in the bucket header that holds the forward pointer to its * first overflow bucket.
10. * * The offset of the field where the length (size) of the partition block is stored * in its header.
11. * * Offset of the field in the bucket header indicating the bucket's element count.
12. * * The length of the header in the partition buffer blocks.
13. * * The array of memory segments that contain the buckets which form the actual hash-table * of hash-codes and pointers to the elements.
14. * * Opens the hash join. This method reads the build-side input and constructs the initial * hash table, gradually spilling partitions that do not fit into memory. * * @throws IOException Thrown, if an I/O problem occurs while spilling a partition.
15. * * The default record width that is used when no width is given. The record width is * used to determine the ratio of the number of memory segments intended for partition * buffers and the number of memory segments in the hash-table structure.
16. * * Constant for the forward pointer, indicating that the pointer is not set.
17. * * Checks, whether the input that is blocked by this iterator, has further elements * available. This method may be used to forecast (for example at the point where a * block is full) whether there will be more data (possibly in another block). * * @return True, if there will be more data, false otherwise.
18. * * The minimum amount of memory that the spilling resettable iterator requires to work.
19. * * Gets the number of elements in the sortable. * * @return The number of elements.
20. for the remaining values, we do a block-nested-loops join
21. match the first values first
22. all input in the block. we don't need to cache the other side
23. * * The fraction of the memory that is dedicated to the spilling resettable iterator, which is used in cases where * the cross product of values with the same key becomes very large.
24. cross the values in the v1 iterator against the current block
25. more data than would fit into one block. we need to wrap the other side in a spilling iterator create spilling iterator on first input
26. get value from the spilling side iterator
27. both sides contain more than one value TODO: Decide which side to spill and which to block!
28. ----------
29. * * Crosses a single value from the second side with N values, all sharing a common key. * Effectively realizes a N:1 match (join). * * @param key The key shared by all values. * @param val1 The value form the 1 side. * @param firstValN The first of the values from the N side. * @param valsN Iterator over remaining N side values. * * @throws RuntimeException Forwards all exceptions thrown by the stub.
30. here, we have a common key! call the match function with the cross product of the values

31. (non-Javadoc) * @see eu.stratosphere.pact.runtime.task.util.MatchTaskIterator#close()
32. first, cross the first value from v1 against all values from the block-resettable iterator that way, we also determine, if there would be any further blocks, which tells us if we need the spilling iterator at all NOTE: Here we still have the first V1 value in the copier!
33. as long as there are blocks from the blocked input
34. * * Crosses a single value from the first input with N values, all sharing a common key. * Effectively realizes a 1:N match (join). * * @param key The key shared by all values. * @param val1 The value form the 1 side. * @param firstValN The first of the values from the N side. * @param valsN Iterator over remaining N side values. * * @throws RuntimeException Forwards all exceptions thrown by the stub.
35. get a value copy
36. (non-Javadoc) * @see eu.stratosphere.pact.runtime.task.util.MatchTaskIterator#abort()
37. * * The log used by this iterator to log messages.
38. * * Calls the <code>MatchStub#match()</code> method for all two key-value pairs that share the same key and come * from different inputs. The output of the <code>match()</code> method is forwarded. * <p> * This method first zig-zags between the two sorted inputs in order to find a common * key, and then calls the match stub with the cross product of the values. * * @throws IOException Thrown, when the reading from the inputs causes an I/O error. * * @see eu.stratosphere.pact.runtime.task.util.MatchTaskIterator#callWithNextKey()
39. ==========
40. spilling is required if the blocked input has data beyond the current block. in that case, create the spilling iterator
41. (non-Javadoc) * @see eu.stratosphere.pact.runtime.task.util.MatchTaskIterator#open()
42. utility classes to make deep copies by serializing and de-serializing the data types
43. * * Utility class that turns a standard {@link java.util.Iterator} for key/value pairs into a * {@link LastRepeatableIterator}.
44. ----------
45. * * Utility function that composes a string for logging purposes. The string includes the given message and * the index of the task in its task group together with the number of tasks in the task group. * * @param message The main message for the log. * @return The string ready for logging.
46. log a final end message
47. the iterator that does the actual matching
48. * * Interface of an iterator that performs the logic of a match task. The iterator follows the * open/next/close principle. The next logic here calls the match stub with all * value pairs that share the same key. * * @author Erik Nijkamp * @author Stephan Ewen
49. the reader who's input is encapsulated in the iterator
50. (non-Javadoc) * @see java.util.Iterator#next()
51. (non-Javadoc) * @see java.util.Iterator#remove()
52. (non-Javadoc) * @see java.util.Iterator#hasNext()
53. allocate the memory for the HashTable
54. * This test is basically identical to the "testSpillingHashJoinWithMassiveCollisions" test, only that the number * of repeated values (causing bucket collisions) are large enough to make sure that their target partition no longer * fits into memory by itself and needs to be repartitioned in the recursion again.
55. the following two values are known to have a hash-code collision on the first recursion level. we use them to make sure one partition grows over-proportionally large
56. create a probe input that gives 10 million pairs with 10 values sharing a key
57. create a build input that gives 3 million pairs with 3 values sharing the same key, plus 400k pairs with two colliding keys
58. create the I/O access for spilling
59. shut down I/O manager and Memory Manager and verify the correct shutdown
60. ----------
61. expected

git_commits:

1. **summary:** Finished Hash Join, Adopted Match Task Iterators, ensured SortMatch will stay in-memory for N:M, if possible, cleaned iterator interfaces.
   message: Finished Hash Join, Adopted Match Task Iterators, ensured SortMatch will stay in-memory for N:M, if possible, cleaned iterator interfaces.

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

jira_issues_comments: